Machine Learning II

Martha White

April 3, 2025

Table of Contents

Notation Reference

Set notation

$\mathcal{X}$ A generic set of values. For example, $\mathcal{X} = \{0, 1\}$ is the set containing only 0 and 1, $\mathcal{X} = [0, 1]$ is the interval from 0 to 1 and $\mathcal{X} = \mathbb{R}$ is the set of real numbers. Depending on occasion, symbols such as A, B, Ω, and others will also be used as sets.

$\mathcal{P}(\mathcal{X})$ The power set of $\mathcal{X}$, a set containing all possible subsets of $\mathcal{X}$.

$[a, b]$ Closed interval with $a < b$, including both a and b.

(a, b) Open interval with $a < b$, with neither a nor b in the set.

$(a, b]$ Open-closed interval with $a < b$, including b but not a.

$[a, b)$ Closed-open interval with $a < b$, including a but not b.

Vector and matrix notation

x Unbold lowercase variables are generally scalars. However, when $x \in \mathcal{X}$, where $\mathcal{X}$ is not specified, x may imdicate a vector, a structured object such as graph, etc.

$\mathbf{x}$ Bold lowercase variables are vectors. By default, vectors are column vectors.

$\mathbf{X}$ Bold uppercase variables are matrices. This looks like a multivariate random variable, $\boldsymbol{X}$, but the random variable is italicized. It will often be clear from context when this is a multivariate random variable and when it is a matrix.

$\mathbf{X}^\top$ The transpose of the matrix. For two matrices $\mathbf{A}$ and $\mathbf{B}$, it holds that

$$(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top.$$

An $n \times d$ matrix consisting of n vectors each of dimension d can be expressed as

$$\mathbf{X} = [\mathbf{x}_1 \, \mathbf{x}_2 \, \ldots \, \mathbf{x}_n]^\top.$$

$\mathbf{X}_{i:}$ The i-th row of the matrix. A row vector.

$\mathbf{X}_{:j}$ The j-th column of the matrix. A column vector.

Tuples, vectors, and sequences

$(x_1, x_2, \ldots, x_d)$ A tuple; i.e., an ordered list of d elements. When $(x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$, the tuple will be treated as a column vector $\mathbf{x} = [x_1\ x_2\ \ldots\ x_d]^\top$.

$a_1, \ldots, a_m$ A sequence of m items. Index variables over these sequences are usually the variables i, j, or k. For example, $\sum_{i=1}^m a_i$ or, if each $\mathbf{a}_i$ is a vector of dimension d, then the double index $\sum_{i=1}^m \sum_{j=1}^d a_{ij}$.

Function notation

$f : \mathcal{X} \to \mathcal{Y}$ The function is defined on domain $\mathcal{X}$ to co-domain $\mathcal{Y}$, taking values $x \in \mathcal{X}$ and sending them to $f(x) \in \mathcal{Y}$.

$\frac{df}{dx}(x)$ The derivative of a function at $x \in \mathcal{X}$, where $f : \mathcal{X} \to \mathbb{R}$ for $\mathcal{X} \subset \mathbb{R}$.

$\nabla f(\mathbf{x})$ The gradient of a function at $\mathbf{x} \in \mathcal{X}$, where $f : \mathcal{X} \to \mathbb{R}$ for $\mathcal{X} \subset \mathbb{R}^d$. It holds that

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_d} \right).$$

$\ell : \mathbb{R}^d \to \mathbb{R}$ A loss function indicating the error in prediction incurred by the given weights, $\ell(\mathbf{w})$. If subscripted, ℓ_i typically indicates the loss on the ith instance, with $\ell(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \ell_i(\mathbf{w})$ for n instances.

$c : \mathbb{R}^d \to \mathbb{R}$ A generic objective function, that we want to minimize, for the learned variable $\mathbf{w}$. This could be, for example, a loss plus a regularizer.

Random variables and probabilities

X A univariate random variable is written in uppercase.

$\mathcal{X}$ The space of values for the random variable.

x Lowercase variable is an instance or outcome, $x \in \mathcal{X}$.

$\boldsymbol{X}$ A multivariate random variable is written bold uppercase.

$\boldsymbol{x}$ Lowercase bold variable is a multivariate instance. In particular cases, when the variable value is treated as a vector, we will use $\mathbf{x}$.

$\mathcal{N}(\mu, \sigma^2)$ A univariate Gaussian distribution, with parameters μ, σ^2.

$\sim$ indicates that a variable is distributed as e.g., $X \sim \mathcal{N}(\mu, \sigma^2)$.

Parameters and estimation

$\mathcal{D}$ A data set, typically composed of n elements of multivariate inputs $\mathbf{X} \in \mathbb{R}^{n \times d}$ and univariate outputs $\mathbf{y} \in \mathbb{R}^n$ or multivariate outputs $\mathbf{Y} \in \mathbb{R}^{n \times m}$. The data set will also be referred to as a set of indexed tuples; i.e., $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ..., (\boldsymbol{x}_n, y_n)\}$.

$\mathcal{F}$ The *function class* or *hypothesis space*. Our learning algorithms will be restricted implicitly to selecting a function from this set. For example, in linear regression, our function class is $\mathcal{F} = \{f : \mathbb{R}^d \to \mathbb{R} \mid f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} for some \mathbf{w} \in \mathbb{R}^d\}$.

$\boldsymbol{\omega}$ The true parameters for the (generalized) linear regression and classification models, typically with $\boldsymbol{\omega} \in \mathbb{R}^d$.

$\mathbf{w}$ The approximated parameters for the (generalized) linear regression and classification models, typically with $\mathbf{w} \in \mathbb{R}^d$. When discussing $\mathbf{w}$ as the maximum likelihood solution on some data, we write $\mathbf{w}_{\text{ML}}(\mathcal{D})$, to indicate that the variability arises from $\mathcal{D}$.

$\max_{a \in \mathcal{B}} c(a)$ The maximum value of a function c across values a in a set $\mathcal{B}$.

$\text{argmax}_{a \in \mathcal{B}} c(a)$ The item a in set $\mathcal{B}$ that produces the maximum value $c(a)$.

Norms

$\|\mathbf{x}\|$ A norm on $\mathbf{x}$.

$\|\mathbf{x}\|_2$ The ℓ_2 norm on a vector, $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^d x_i^2}$. This norm gives the Euclidean distance from the origin of the coordinate system to $\mathbf{x}$; that is, it is the length of vector $\mathbf{x}$.

$\|\mathbf{x}\|_2^2$ The squared ℓ_2 norm on a vector, $\|\mathbf{x}\|_2^2 = \sum_{i=1}^d x_i^2$.

$\|\mathbf{x}\|_p$ The general ℓ_p norm on a vector, $\|\mathbf{x}\|_p = (\sum_{i=1}^d |x|_i^p)^{1/p}$.

Useful formulas and rules

$$\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$$

$$\log\left(x^y\right) = y\log(x)$$

$$\sum_{i=1}^{m} a_i \int_{\mathcal{X}} f_i(x)p(x)dx = \int_{\mathcal{X}} \sum_{i=1}^{m} a_i f_i(x)p(x)dx \quad \triangleright \text{ Can bring sum into integral}$$

$$\frac{d}{dx}\int_{\mathcal{X}} f(x)p(x)dx = \int_{\mathcal{X}} \frac{d}{dx} f(x)p(x)dx \quad \triangleright \text{ Can (almost always) bring derivative into integral}$$

Chapter 1

Introduction to Machine Learning II

These notes presume you have already learned about the basics of machine learning. This includes the following core concepts: probabilistic underpinnings, estimators and formalizing objectives to obtain those estimators (MLE and MAP), evaluating confidence in an estimator, bias-variance, generalization and overfitting, regularization and basic optimization strategies and algorithms such as linear regression. Ideally, you know all the topics from the Machine Learning I notes [31]. The ML I notes are quite short, and could be read now before diving deeply into these ML II notes. The ML I notes are also full of exercises that could be a good way to brush up on your knowledge. In this chapter, we provide a very brief overview of ML I in Section 1.1.

These notes expand on these basics from ML I, primarily by revisiting and extending each of the concepts and by introducing a key concept not yet covered: data representation. You will see more complex distributions and maximum likelihood applied to those distributions (e.g., mixture models and expectation-maximization). With these more complex distributions, it becomes more sensible to discuss generative models, not just predictors; this course will cover both more, as opposed to the basics which focused primarily on prediction. You will also see more advanced ways to learn nonlinear predictors, beyond simply using polynomial features, including neural networks and kernel (similarity) features. You will see other regularization approaches and more advanced optimization strategies (e.g., proximal methods for ℓ_1 regularization). We will cover Bayesian methods and uncertainty estimation generalization for a broader class of models.

Throughout, we will see the central concept of data representation. Many methods rely on re-representing inputs, to facilitate modeling. We already touched on this lightly with polynomial features. Inputs were transformed into polynomial features, to make it easy to learn nonlinear predictors using linear regression. We will discuss how high-dimensional representations can facilitate learning linear regressors and classifiers. We will also discuss how identifying a (compact) set of latent factors provides a data representation that facilitates learning generative models (distributions) and handling missing data.

1.1 A Very Brief Refresher of the Basics of Machine Learning

In this section we do a whirlwind refresher of the concepts and terminology learned in the basics of machine learning. Our primary goal was to learn a prediction function $f_{\mathbf{w}} : \mathcal{X} \to \mathcal{Y}$, parameterized by a vector of weights $\mathbf{w} \in \mathbb{R}^k$. This prediction function inputs a vector of observations $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^d$ and outputs a prediction $\hat{y} \in \mathcal{Y}$. If $\mathcal{Y}$ is a discrete, unordered set, like $\mathcal{Y} = \{\texttt{giraffe}, \texttt{hippo}, \texttt{ostrich}\}$, then we call the problem of finding f a **classification**

problem. If $\mathcal{Y}$ is continuous, then we say it is a **regression** problem.[1]

We discussed (a) how to learn such a function and (b) how to evaluate if that function is good. To learn the function, we needed a clear criterion (objective function) to optimize. We discussed that the ultimate goal is to find a function f with low **expected cost**, $\mathbb{E}[\text{cost}(f(\boldsymbol{X}), Y)]$, which we later called the **generalization error** of f. This cost was different for different problems. For regression, we used $\text{cost}(f(\mathbf{x}), y) = (f(\mathbf{x}) - y)^2$ and for classification we used the 0-1 cost

$$\text{cost}(\hat{y}, y) = \begin{cases} 0 & \text{when } y = \hat{y} \\ \\ 1 & \text{when } y \neq \hat{y} \end{cases}$$

We found that these choices for costs implied that the optimal predictor for regression is $f^*(\mathbf{x}) = \mathbb{E}[Y|\mathbf{x}]$ and for classification is $f^*(\mathbf{x}) = \text{argmax}_{y \in \mathcal{Y}} p(y|\mathbf{x})$. This motivated estimating $p(y|\mathbf{x})$, or the mean of this distribution $\mathbb{E}[Y|\mathbf{x}]$, using data.

Formalizing the problem was fun, but now we have the hard part of estimating these unknown quantities. We know $f^*(\mathbf{x}) = \mathbb{E}[Y|\mathbf{x}]$ for regression, but we don't have $\mathbb{E}[Y|\mathbf{x}]$! Instead, we only have a dataset of samples $\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $(\mathbf{x}_i, y_i) \sim p$ where $p(\mathbf{x}, y) = p(y|\mathbf{x})p(\mathbf{x})$. This dataset is a poor proxy, but we will have to make do. The parameters $\mathbf{w}$ for the function we learn are actually parameters for the distribution of $p(y|\mathbf{x})$. Therefore, we decided to find parameters that were the most likely, given the data: the MAP objective.

For regression we modeled the conditional distribution as a Gaussian with fixed variance σ^2, written as $p(y|\mathbf{x}) = \mathcal{N}(f_{\mathbf{w}}(\mathbf{x}), \sigma^2)$. The data gives us clues about the true f^* that defines the conditional mean. We want to pick the $f_{\mathbf{w}}$ that is the most likely, given this evidence. In other words, the **MAP objective** is

$$\begin{aligned} \underset{\mathbf{w} \in \mathbb{R}^k}{\text{argmax}}\, p(\mathbf{w}|\mathcal{D}) &= \underset{\mathbf{w} \in \mathbb{R}^k}{\text{argmax}}\, p(\mathcal{D}|\mathbf{w})p(\mathbf{w}) \\ &= \underset{\mathbf{w} \in \mathbb{R}^k}{\text{argmax}} \sum_{i=1}^n \ln p(y_i|\mathbf{x}_i, \mathbf{w}) + \ln p(\mathbf{w}) \\ &= \underset{\mathbf{w} \in \mathbb{R}^k}{\text{argmin}} - \sum_{i=1}^n \ln p(y_i|\mathbf{x}_i, \mathbf{w}) - \ln p(\mathbf{w}) \end{aligned}$$

where the first step drops constants, the second uses monotonicity of log and the third uses the equivalence between maximizing a function and minimizing the negative of that function. The term $p(\mathcal{D}|\mathbf{w})$ is called the **likelihood**, the term $p(\mathbf{w})$ the **prior** (before seeing evidence) and the term $p(\mathbf{w}|\mathcal{D})$ the **posterior** (after seeing evidence).

The prior allows us to inject our own knowledge, and so constrain the space of possible solutions. We considered a Gaussian prior on $\mathbf{w}$, to encourage the weights to be near zero. We did so because we concluded large weights can indicate **overfitting**. Overfitting occurs when the learned function $f_{\mathbf{w}}$ specializes to the training set, at the cost of generalization

[1] If $\mathcal{Y}$ is discrete but ordered, then sometimes this is modeled as an ordinal regression problem. An example of an ordinal regression problem is one where the goal is to predict the number of injuries in a day. Then $\mathcal{Y} = \{0, 1, 2, 3, 4, \ldots\}$, and the set is ordered: 4 injuries is more similar to 5 injuries, than to 100 injuries. We did not talk about ordinal regression before, but when we talk about generalized linear models, we will see how Poisson regression can be used for this ordinal regression problem.

performance. We saw that for very small training sets, with polynomial regression, we could almost perfectly fit the training dataset, but the resulting function had very poor generalization error. The true underlying function was actually simpler, and the additional degrees of freedom from the polynomial was used to fit noise (from variance σ^2 in $Y|\mathbf{x}$) rather than identify the pattern $\mathbb{E}[Y|\mathbf{x}]$. This addition of a Gaussian prior corresponded to ℓ_2 **regularization**.

In some cases, we may not want to constrain solutions with a prior, potentially because we simply do not know what prior to pick. In that case, we may want to maximize the likelihood. As we discussed, conceptually this is like picking a uniform prior in MAP. This **maximum likelihood (MLE)** objective—equivalently negative log likelihood objective—is

$$\operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^k} -\sum_{i=1}^{n} \ln p(y_i|\mathbf{x}_i, \mathbf{w}).$$

After finding this function $f_{\mathbf{w}}$, using MAP or MLE, we want to evaluate if it is good. The gold standard is the generalization error of $f_{\mathbf{w}}$: $\mathbb{E}[(f_{\mathbf{w}}(\boldsymbol{X}) - Y)^2]$. However, again we cannot directly compute this, as it is an expectation over all possible pairs $(\mathbf{x}, y)$. Instead, we can use data to estimate it and we can reason conceptually (or theoretically) about whether we should expect $f_{\mathbf{w}}$ to have good generalization error.

To estimate the generalization error with data, we can use a (hold-out) **test set**. This means that we take the dataset and split it into a training set (say 80% of the data) $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ and use the rest as a test set $\mathcal{D}_{\text{test}} = \{(\mathbf{x}_i, y_i)\}_{i=n+1}^{n+m}$. This ensures that the test set is independent of the training set: they have independent samples of pairs $(\mathbf{x}, y)$. We can then use a sample average estimate of the generalization error using

$$\text{GE}(f_{\mathbf{w}}) \approx \hat{\text{GE}}(f_{\mathbf{w}}) \stackrel{\text{def}}{=} \frac{1}{m} \sum_{i=n+1}^{n+m} \text{cost}(f_{\mathbf{w}}(\mathbf{x}_i), y_i)$$

It is not enough to simply use this sample average estimate, we also want a notion of confidence in this estimate. In other words, we want a less vague relationship between $\text{GE}(f_{\mathbf{w}})$ and $\hat{\text{GE}}(f_{\mathbf{w}})$ than our approximately equals to symbol $\approx$. We obtained a more precise relationship using a confidence interval around $\hat{\text{GE}}(f_{\mathbf{w}})$. When reporting our estimate of generalization error, therefore, we provide the interval $[\hat{\text{GE}}(f_{\mathbf{w}}) - \epsilon, \hat{\text{GE}}(f_{\mathbf{w}}) + \epsilon]$ for an ϵ that gives the width of the interval, determined by distributional assumptions and the level of confidence required $1 - \delta$. For example, if we assumed errors $(f_{\mathbf{w}}(\boldsymbol{X}_i) - Y_i)^2$ are Gaussian distributed with unknown mean and variance, then we modeled $\hat{\text{GE}}(f_{\mathbf{w}})$ using a Student t-distribution. The resulting confidence interval, say if $\delta = 0.05$ and $m = 10$, is given by $\epsilon = \frac{2.26}{\sqrt{m}} \times S_m$ for S_m the unbiased sample standard deviation of the errors.

In addition to empirical measures, we also reasoned about whether we should expect $f_{\mathbf{w}}$ to generalize well. For example, we reasoned that if $f_{\mathbf{w}}$ is a 9th-order polynomial but we only have three data points, then likely we will not generalize well due to overfitting. This conceptual reasoning is about the **bias** and **variance** of different function classes and algorithms to find these functions. We reasoned that in some cases it was worthwhile to incur some bias to reduce variance. Ultimately, we combine conceptual reasoning to select the set of models we consider (e.g., low or high order polynomials, with or without regularization) with empirical estimates of generalization error to deploy learned functions.

A theme underlying the entire course is the notion of a probabilistic formulation to quantify **uncertainty in our estimators**. We have sensible ways to obtain sample average

estimators, or parameters of distributions like the variance, or the parameters for a function that give conditional distributions. But, we may also want to know the uncertainty in our estimates. For sample averages, we use concentration inequalities to get confidence intervals to reflect this uncertainty. For our parameterized functions, such as in linear regression, we use a Bayesian approach to obtain **credible intervals** over both the parameters and the predictions.

All of the above equally applies to classification with **logistic regression**. We used the same analysis to reason about (a) optimal predictors for classification, (b) the resulting MAP or MLE optimization problem to approximate the predictor and (c) conceptual and empirical strategies to evaluating generalization error of the learned functions.

Finally, an important theme throughout the course was **optimization algorithms** strategies to actually solve the optimization problems. We spent a lot of time formalizing and understanding the goals of learning, as described above, but eventually we have to actually implement it on a computer. We discussed **gradient descent** to solve our smooth, continuous optimization problems, and the importance of step-size selection. We then discussed the clever generalization to **mini-batch stochastic gradient** descent, which similarly reaches *local minima* but with less computation.

1.2 Generative Models and Predictors

There are two typical goals in machine learning: learning a generative model and learning a predictor. Many of the concepts are similar between the two, because they both rely on estimating parameters for a distribution. When we learn the distribution $p(x)$, we are learning a **generative model**. When we learned the conditional distribution $p(y|x)$, in regression and classification, we were learning a predictor.

This distinction, however, is not quite crisp. The real defining difference is how we will use the models we learn. We typically learn a generative model, to allow us to sample—or generate—items. For example, we might learn a generative model of faces, to allow us to generate new images of faces. The model produces hypothetical images, rather than making predictions. This means that we could in fact learn a conditional generative model, $p(y|x)$, where different faces y could be sampled depending on context x. For example, we may want to set $x = \textit{narrow}$ to sample only narrow faces. This face distribution $p(y|x)$ is much more complex than the distributions we considered for regression and classification.

In regression, on the other hand, we are primarily interested in statistics of the distribution that enable us to make predictions. The distributions themselves can be quite simple. For example, for linear regression, we assumed $p(y|x)$ is Gaussian with a fixed variance across all x. Our predictions usually correspond to $\mathbb{E}[Y|x]$, though it can also be sensible to use other statistics like the median$(Y|x)$. The power and complexity is in the features x, for which we are not trying to estimate the distribution. Using the same example as above, x could be features and y might be a label such as *narrow* or *not narrow*. The distribution $p(y|x)$ is a conditional Bernoulli, which is simple even though x is complex. This contrasts the above where our goal was to learn the distribution over the complex object, namely over the faces.

In summary, the primary differences are in (a) how we use the distribution and (b) the complexity of the distribution. Both generative models and prediction models will rely on similar nonlinear modelling tools, like neural networks and kernels. But, on top of those

approaches, the strategies will look different due to these two differences. The algorithms we use to learn generative models will typically be slightly more complex, to learn these more complex distributions. Moreover, we will have to evaluate the generative models differently, since their use case is different. In the basics, we only discussed evaluating prediction models; now, we will also discuss how to evaluate generative models.

1.3 Relationship to Statistics and Probability Theory

Machine learning is based on tools from statistics and from probability. You may wonder why much of machine learning comes out of computer science departments, rather than statistics. The answer is primarily due to a difference in focus. Statistics historically focused on understanding data. Consequently, *inference* more so than prediction is critical. For prediction, we want to learn a function f on inputs x that give us accurate predictions of y. For inference, we instead want to *understand the relationship* between the inputs x and targets y. Which inputs are most correlated with the target? And is an input variable positively or negatively correlated?

Simple and interpretable models are useful for inference. For example, linear models can make the relationship between an input variable x_j and y more clear. If the coefficient is large, x_j may be an important predictor for y. Further, if it is negative, the correlation is negative. Nonlinear models can complicate understanding the relationships.

Conversely, for prediction, with the advent of more and more data, it is feasible to learn larger and more complex models. These more complex models are straightforward to use for prediction, because the interpretability is not as relevant. Our focus in this course is on using models—namely to obtain predictive models and generative distributions—rather than inference.

1.4 The Blessing and Curse of Dimensionality

This course focuses on high-dimensional vectors of features (representations). High dimensionality can be both a blessing and a curse. The term curse of dimensionality actually arose from solving high-dimensional dynamic programming problems. Each added dimension can cause an exponential growth in the search space for such discrete optimization problems. In machine learning, we typically focus on continuous variables, rather than discrete, but similar difficulties with increasing dimension do arise. The curse in machine learning usually takes two forms: the requirements on the number of samples and the fact that similarities breakdown in high dimensions.

The first issue arises due to how quickly (exponentially) the volume of a d-dimensional space grows. With more features—higher dimensionality—we require more samples to identify the correct model. Imagine we have a $[0, 1]^d$ space and we want to cover our input space with a grid. Say, we want to reason about seeing a point $\mathbf{x} \in \mathbb{R}^d$ for each region of the input space, at a spacing of 0.1 (10 per dimension). If $d = 1$ and we only have an interval, we only need 10 points to cover the space at a resolution of 0.1. If we have $d = 2$, then we need $10^2 = 100$ points to cover at the same resolution. In general, we need 10^d, which means that for $d = 3$ we need 1000 points, for $d = 4$ we need $10,000$ and quickly this balloons out of proportion. We do not actually need to fully cover our space in machine learning, but

we can see that in higher dimensions a small number of samples barely covers the space at all. We see a minuscule fraction of the possible input space.

The second issue arises from counterintuitive concentration phenomena—discussed more later—that occur in high-dimensions. Namely, the volume of a high-dimensional ball or cube is concentrated near its surface, rather than the interior. One implication of this phenomena is that, in high-dimensions, the distance between a vector and any other random vector is likely to be close to the average distance. More specifically, give a randomly sampled dataset of points, the distance to the closest and nearest neighbors for a given point in that dataset approaches one [7]. This means that relying on similarities between feature vectors—as we do in certain learning algorithms—can be ineffective.

This phenomena, however, also provides a blessing. In high-dimensions, data becomes *separable*. That is, if we want to classify individual points differently, then projecting into higher dimensions quickly makes this feasible. For example, when using a feature expansion, such as with prototype representations in Chapter 8, we can find a linear classifier in this new space that perfectly separates the two classes.

More generally, the loaded terminology of *curse* and *blessing* detracts from the fact that different situations warrant different strategies. The properties of high-dimensional spaces need to be considered carefully, to avoid known failures. But, there is no doubt that learning high-dimensional data representations is an important way forward for prediction. More complex models, with many many parameters, can significantly improve performance and do not suffer from some of the overfitting problems that intuitively might have been expected. Leveraging these nice properties, particularly given by neural networks, has generated a flurry of work in machine learning.

1.5 Matrix Methods

The basics course avoided matrix methods. Comfort with linear algebra takes time, and is not strictly needed to implement some of the most commonly used machine learning methods. As you learned, many of the estimation approaches can use (stochastic) gradient descent. Our focus was on understanding how stochastic gradient descent can reach stationary points, the role of the stepsize in doing so, and why it is that we would want to reach stationary points. This procedure is quite generic, requiring simply that we can compute the gradient of our specified objective. For example, for linear regression, the gradient for a single sample consisted of the (signed) error of our prediction times the feature vector for that input.

In some cases, however, we can find closed form solutions using matrix methods. In linear regression, for example, we could have computed the solution by iterating over the entire dataset only once—instead of for multiple epochs—and computing a matrix inverse. We can express the linear regression solution as the inverse of a matrix times a vector. This expression allows us to better understand the properties of the linear regression solution, as we will see in this course.

Similarly, matrix approaches can elucidate what is learned under certain data representations. For the latent variable methods we consider, formalizing the problem using matrices will help us understand the latent factors extracted.

For this course, I expect you to recall a few basics, summarized in this section.

1.5.1 Matrix multiplication

Recall that a $m \times n$ matrix $\mathbf{A}$ is a two-dimensional array with m rows and n columns.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \dots & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \dots \\ \mathbf{a}_m \end{bmatrix}$$

where $\mathbf{a}_i \stackrel{\text{def}}{=} [a_{i1}, a_{i2}, \dots, a_{in}]$ is a vector corresponding to the ith row. The symbol is bold, to indicate it is a vector. Sometimes, this row vector is written using the notation $\mathbf{A}_{i:}$ where the colon indicates the entire row from 1 to n is used. Similarly, to reference a column, we use $\mathbf{A}_{:j}$. When you multiply a vector $\mathbf{x} \in \mathbb{R}^n$ with this matrix, the matrix-vector dot product corresponds to doing a dot product between $\mathbf{x}$ and each row of $\mathbf{A}$:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{x} \rangle \\ \langle \mathbf{a}_2, \mathbf{x} \rangle \\ \dots \\ \langle \mathbf{a}_m, \mathbf{x} \rangle \end{bmatrix} = \begin{bmatrix} \langle \mathbf{A}_{1:}, \mathbf{x} \rangle \\ \langle \mathbf{A}_{2:}, \mathbf{x} \rangle \\ \dots \\ \langle \mathbf{A}_{m:}, \mathbf{x} \rangle \end{bmatrix} \in \mathbb{R}^m$$

The resulting vector is the same dimension as the number of rows of $\mathbf{A}$. More generally, if we do a matrix-matrix product with another matrix $\mathbf{B} \in \mathbb{R}^{n \times k}$

$$\mathbf{A}\mathbf{B} = [\mathbf{A}\mathbf{B}_{:,1}, \mathbf{A}\mathbf{B}_{:,2}, \dots, \mathbf{A}\mathbf{B}_{:,k}] = \begin{bmatrix} \mathbf{A}_{1,:}\mathbf{B}_{:,1} & \mathbf{A}_{1,:}\mathbf{B}_{:,2} & \dots & \mathbf{A}_{1,:}\mathbf{B}_{:,k} \\ \mathbf{A}_{2,:}\mathbf{B}_{:,1} & \mathbf{A}_{2,:}\mathbf{B}_{:,2} & \dots & \mathbf{A}_{2,:}\mathbf{B}_{:,k} \\ & \dots & & \\ \mathbf{A}_{m,:}\mathbf{B}_{:,1} & \mathbf{A}_{m,:}\mathbf{B}_{:,2} & \dots & \mathbf{A}_{m,:}\mathbf{B}_{:,k} \end{bmatrix} \in \mathbb{R}^{m \times k}$$

To multiply these matrices, the inner dimension n has to match: $\mathbf{A}$ is $m \times n$ and $\mathbf{B}$ is $n \times k$. The resulting matrix has the same number of rows as $\mathbf{A}$ and number of columns as $\mathbf{B}$. It is a useful sanity check in derivations, to see if you have made a mistake, to check if all the dimensions match. If you find yourself multiplying two matrices with different inner dimensions, then something went wrong.

A useful rule is that $(\mathbf{A}\mathbf{B})^\top = \mathbf{B}^\top \mathbf{A}^\top$. Recall that the transpose involves flipping the matrix around its diagonal. Namely,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \dots & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \qquad \mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ & \dots & & \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}$$

If $\mathbf{A} \in \mathbb{R}^{m \times n}$, then $\mathbf{A}^\top \in \mathbb{R}^{n \times m}$.

Exercise 1: Prove that $(\mathbf{A}\mathbf{B})^\top = \mathbf{B}^\top \mathbf{A}^\top$. You can do so by computing both the rhs and lhs and ensuring they are the same. □

1.5.2 Matrix Inverse and Eigenvalue Decomposition

We will also use matrix inverses. Recall that for a scalar number a, it's inverse a^{-1} is $1/a$, because $aa^{-1} = 1$ (which is the definition of an inverse). For a diagonal matrix $\mathbf{A}$, its inverse

is similarly straightforward:

$$\mathbf{A} = \begin{bmatrix} a_1 & 0 & \dots & 0 & 0 \\ 0 & a_2 & 0 & \dots & 0 \\ & \dots & & & \\ 0 & 0 & \dots & 0 & a_d \end{bmatrix} \qquad \mathbf{A}^{-1} = \begin{bmatrix} 1/a_1 & 0 & \dots & 0 & 0 \\ 0 & 1/a_2 & 0 & \dots & 0 \\ & \dots & & & \\ 0 & 0 & \dots & 0 & 1/a_d \end{bmatrix}$$

where you can verify that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ for identity matrix $\mathbf{I}$ that has 1s on the diagonal

$$\mathbf{I} \stackrel{\text{def}}{=} \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 \\ & \dots & & & \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

We will also use determinants and matrix decompositions—including the *eigenvalue decomposition* and *singular value decomposition*—as well as the rank of a matrix. Every matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ has a singular value decomposition $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ where $\mathbf{U} = [\mathbf{u}_1, \dots, \mathbf{u}_m] \in \mathbb{R}^{m \times m}$ is the orthonormal matrix composed of the left singular vectors, $\mathbf{V} = [\mathbf{v}_1, \dots, \mathbf{v}_n] \in \mathbb{R}^{n \times n}$ is the orthonormal matrix composed of the right singular vectors. An orthonormal matrix $\mathbf{U}$ is square matrix that satisfies $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$ and $\mathbf{U}\mathbf{U}^\top = \mathbf{I}$. The matrix of nonnegative singular values $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ is a diagonal matrix with zero padding in the dimension that is larger. For example, consider the case where $m > n$. The diagonal entries in $\mathbf{\Sigma}$ are the singular values, which we typically order in descending order $\sigma_1, \sigma_2, \dots, \sigma_n$, giving

$$\mathbf{\Sigma} \stackrel{\text{def}}{=} \begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ & & \vdots & & \\ 0 & 0 & \dots & 0 & \sigma_n \\ 0 & 0 & \dots & 0 & 0 \\ & & \vdots & (m-n) \text{ rows} & \text{of zeros} \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{\Sigma}_n \\ \mathbf{0} \end{bmatrix} \quad \text{where } \mathbf{\Sigma}_n \stackrel{\text{def}}{=} \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ & & \vdots & \\ 0 & 0 & \dots & \sigma_n \end{bmatrix}.$$

Any matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ can be decomposed into its singular value decomposition, because any linear transformation can be decomposed into a rotation (multiplication by $\mathbf{V}^\top$), followed by a scaling (multiplication by $\mathbf{\Sigma}$), followed again by a rotation (multiplication by $\mathbf{U}$).

Sometimes we use the *thin SVD*, where drop the columns of $\mathbf{U}$ and rows of $\mathbf{V}$ corresponding to the zero parts of $\mathbf{\Sigma}$. Let us call these $\mathbf{U}_k \in \mathbb{R}^{m \times k}$ and $\mathbf{V}_k \in \mathbb{R}^{k \times n}$, the first k columns and rows respectively where $k = \min(m, n)$. For the above, where we consider $m > n$, we have $k = n$. We can do this because the zero parts of $\mathbf{\Sigma}$ remove those rows or columns in the multiplication:[2]

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^\top.$$

[2]You might wonder why we do not simply define the SVD this way, since it has less redundant information. The reason is that we need $\mathbf{U}$ and $\mathbf{V}$ to be orthonormal matrices, which means they must be square. The thin versions still satisfy useful orthogonality properties, namely that for $m > n$ with $k = n$, $\mathbf{U}_k^\top\mathbf{U}_k = \mathbf{I}$ but we no longer have $\mathbf{U}_k\mathbf{U}_k^\top = \mathbf{I}$.

In this example, where $m > n$ and $k = n$, we have that $\mathbf{V}_k = \mathbf{V}$. But we still define this $\mathbf{V}_k$ since more generally for the thin SVD we might have $m < n$ and then $\mathbf{V}$ is the one that is made smaller.

This decomposition simplifies analysis of the properties of a matrix. For example, the number of non-zero singular values constitutes the rank of $\mathbf{X}$. To see why, assume $\sigma_n = 0$, and $\sigma_{n-1} > 0$, meaning $\mathbf{X}$ has rank $n-1$. Take any vector $\mathbf{w} \in \mathbb{R}^n$, and consider $\mathbf{Xw}$. We can write this product as

$$\mathbf{U\Sigma V}^\top \mathbf{w} = \mathbf{U\Sigma}\tilde{\mathbf{w}} \qquad \text{for } \tilde{\mathbf{w}} = \mathbf{V}^\top \mathbf{w}.$$

The product $\mathbf{\Sigma}\tilde{\mathbf{w}}$ sets the last dimension of $\tilde{\mathbf{w}}$ to zero, effectively removing that dimension and so projecting $\tilde{\mathbf{w}}$ into a lower-dimensional $(n-1)$ space. Then it rotates that projected vector afterwards, using $\mathbf{U}$, but cannot undo the projection into a lower-dimensional space. Therefore, $\mathbf{Xw}$ can only produce $\hat{\mathbf{y}} = \mathbf{Xw}$ that lie in a $n-1$-dimensional plane.

The *eigenvalue decomposition* is similar to the SVD, but is only defined for square matrices $\mathbf{A} \in \mathbb{R}^{m \times m}$. We will further only consider the eigenvalue decomposition of symmetric matrices—since that is all that we need for these notes—which slightly simplifies the decomposition. The eigenvalue decomposition of a square, symmetric matrix is $\mathbf{A} = \mathbf{U\Lambda U}^\top$ for orthonormal matrix $\mathbf{U} \in \mathbb{R}^{m \times m}$ and diagonal matrix $\mathbf{\Lambda} \in \mathbb{R}^{m \times m}$, where now the entries can be both positive and negative. The number of non-zero elements again tell us the rank of the matrix.

The eigenvalue decomposition—and the SVD, for that matter—make computation of the inverse straightfoward. Namely, if $\mathbf{A}$ is full rank and so invertible, we get that

$$\begin{aligned}
\mathbf{AA}^{-1} &= (\mathbf{U\Lambda U}^\top)\mathbf{U\Lambda}^{-1}\mathbf{U}^\top \\
&= \mathbf{U\Lambda U}^\top\mathbf{U\Lambda}^{-1}\mathbf{U}^\top && \triangleright\ \mathbf{U}^\top\mathbf{U} = \mathbf{I} \text{ by definition of orthonormal matrices} \\
&= \mathbf{U\Lambda I\Lambda}^{-1}\mathbf{U}^\top && \triangleright\ \text{Identity operator } \mathbf{I} \text{ has no impact} \\
&= \mathbf{U\Lambda\Lambda}^{-1}\mathbf{U}^\top && \triangleright\ \mathbf{\Lambda\Lambda}^{-1} = \mathbf{I} \\
&= \mathbf{UIU}^\top \\
&= \mathbf{UU}^\top && \triangleright\ \mathbf{UU}^\top = \mathbf{I} \text{ by definition of orthonormal matrices} \\
&= \mathbf{I}
\end{aligned}$$

Exercise 2: Go through the same steps above, and prove that given the SVD $\mathbf{A} = \mathbf{U\Sigma V}^\top$ of a full-rank symmetric, square matrix, the matrix $\mathbf{A}^{-1} = \mathbf{V\Sigma}^{-1}\mathbf{U}^\top$ is the inverse of $\mathbf{A}$. □

1.5.3 Basic Rules for Gradients with Vectors and Matrices

For derivatives, there are useful rules that you are familiar with, such as $\frac{d}{dw}aw = a$, $\frac{d}{dw}w^2 = 2w$ and $\frac{d}{dw}e^w = e^w$. We can similarly write down such rules for the multivariate setting, to simplify computation of gradients without having to go resort to computing each partial derivative. Each of the following rules can be verified by computing partial derivatives, with the rules you are used to for the univariate case. We summarize the key rules for this document here; for a more complete reference, see the matrix cookbook [21].

Three useful rules are summarized in Table 1.1. Note to obtain the derivative for the function $f(\mathbf{x}) = \mathbf{x}^\top\mathbf{A}$, one can first obtain the derivative for $f(\mathbf{x})^\top = \mathbf{A}^\top\mathbf{x}$ and then take

$f(\mathbf{x})$	$\frac{\partial f}{\partial \mathbf{x}}$
$\mathbf{x}^\top \mathbf{x}$	$2\mathbf{x}$
$\mathbf{A}\mathbf{x}$	$\mathbf{A}^\top$
$\mathbf{x}^\top \mathbf{A}\mathbf{x}$	$\mathbf{A}\mathbf{x} + \mathbf{A}^\top \mathbf{x}$

Table 1.1: Useful derivative formulas of vectors with respect to vectors. The derivative of vector-valued function $f : \mathbb{R}^{d\times 1} \to \mathbb{R}^{m\times 1}$ with respect to vector $\mathbf{x} \in \mathbb{R}^{d\times 1}$ is an $d \times m$ matrix $\mathbf{M}$ with components $M_{ij} = \partial y_j / \partial x_i$, $i \in \{1, 2, \ldots, d\}$ and $j \in \{1, 2, \ldots, m\}$. A derivative of scalar with respect to a vector, where $m = 1$, is a special case of this situation that results in an $d \times 1$ column vector. Note that in the table, m is not the same for each row. For example, $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{x}$ is a scalar, whereas for a general matrix $\mathbf{A} \in \mathbb{R}^{m\times d}$, $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ is a m-dimensional vector.

its transpose because

$$\big(\nabla f(\mathbf{x})\big)^\top = \nabla\big(f(\mathbf{x})^\top\big).$$

Therefore, because $\nabla(f(\mathbf{x})^\top) = \mathbf{A}$, we get that $\nabla f(\mathbf{x}) = \mathbf{A}^\top$. Note that because of this equivalence in the above equation, we will often drop the brackets and simply write $\nabla f(\mathbf{x})^\top$ instead of $\nabla\big(f(\mathbf{x})^\top\big)$.

Part I

Revisiting Concepts

Chapter 2

Multivariate Probability Concepts

You have already learned about basic probability concepts, including discrete and continuous random variables; pmfs and pdfs; expectations, covariances and moments; independence and conditional independence. In this chapter, we discuss more complex distributions, including multidimensional distributions and mixture models. We additionally discuss *entropy*, which provides another piece of information about a distribution, and *KL divergences* that allow us to compare two distributions.

2.1 Multidimensional distributions

Recall that discrete random variables have probability mass functions (pmfs) and continuous random variables have probability density functions (pdfs). You have already seen several examples of such distributions for univariate random variables (one-dimensional random variables). These included Bernoulli, uniform and Poisson distributions for discrete and uniform, Gaussian, exponential and Gamma distributions for continuous random variables. In this section we briefly revisit these definitions, generally for multivariate random variables—which include the univariate ones as a special case—and provide a few examples.

Let $\boldsymbol{X} = (X_1, X_2, \ldots, X_d)$ be a d-dimensional random variable with vector-valued outcomes $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)$, such that each x_i is chosen from some $\mathcal{X}_i$. The sample space for $\boldsymbol{X}$ is $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots, \times \mathcal{X}_d$. For the discrete case, each $\mathcal{X}_i$ is a countable set, such as $\{1, 2, 3\}$ or the set of integers; for the continuous case, it is a continuous (uncountable) set, such as $[-1, 1]$ or the set of all reals.

For the discrete case, any function $p : \mathcal{X} \to [0, 1]$ is called a probability mass function (pmf) if

$$\sum_{x_1 \in \mathcal{X}_1} \sum_{x_2 \in \mathcal{X}_2} \cdots \sum_{x_d \in \mathcal{X} d} p\left(x_1, x_2, \ldots, x_d\right) = 1.$$

One example of the multidimensional pmf is the *multinomial distribution*, which generalizes the binomial distribution to the case when the number of outcomes in any trial is a positive integer $d \geq 2$. The multinomial distribution is used to model a sequence of n independent and identically distributed (i.i.d.) trials with d outcomes. At each point $(x_1, x_2, \ldots, x_d)$ in the sample space, the multinomial pmf provides the probability that the outcome 1 occurred x_1 times, outcome 2 occurred x_2 times, etc. Of course, $0 \leq x_i \leq n$ for all i and $\sum_{i=1}^{d} x_i = n$. For example, an experiment consisting of n tosses of a fair six-sided die and counting the number of occurrences of each number can be described by a multinomial distribution.

More formally, given the sample space $\mathcal{X} = \{0, 1, \ldots, n\}^d$, the multinomial pmf is defined

as

$$p(x_1, x_2, \ldots, x_d) = \begin{cases} \binom{n}{x_1, x_2, \ldots, x_d} \alpha_1^{x_1} \alpha_2^{x_2} \ldots \alpha_d^{x_d} & \text{if } x_1 + x_2 + \cdots + x_d = n \\ \\ 0 & \text{otherwise} \end{cases}$$

where α_i's are positive coefficients such that $\sum_{i=1}^{d} \alpha_i = 1$. That is, each coefficient α_i gives the probability of outcome i in any trial. The multinomial coefficient

$$\binom{n}{k_1, k_2, \ldots, k_d} = \frac{n!}{k_1! k_2! \cdots k_d!}$$

generalizes the binomial coefficient by enumerating all ways in which one can distribute n balls into d boxes such that the first box contains k_1 balls, the second box k_2 balls, etc.

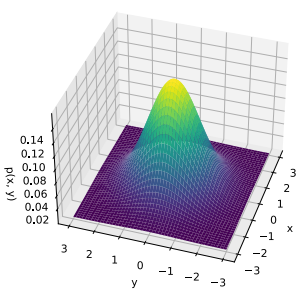
This distribution is useful for us for multi-class classification, where we categorize an item into one of m classes. We will more specifically consider a special case of the multinomial, where $n = 1$. Then d corresponds to the number of classes m and only one class is successful, namely only one x_i is 1 and the remainder are 0. The distribution corresponds to the probability of each class being the correct class (for a given input). Simplifying the above, we have $\mathcal{X} = \{0, 1\}^d$ and pmf

$$p(x_1, x_2, \ldots, x_d) = \begin{cases} \alpha_1^{x_1} \alpha_2^{x_2} \ldots \alpha_d^{x_d} & \text{if } x_1 + x_2 + \cdots + x_d = 1 \\ \\ 0 & \text{otherwise} \end{cases}$$

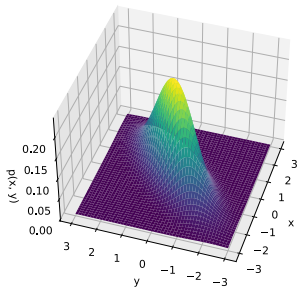
When $d = 2$, this reduces to the Bernoulli distribution. The above pmf for $d = 2$ is $\alpha_1^{x_1} \alpha_2^{x_2}$, where $x_1 + x_2 = 1$ and $\alpha_1 + \alpha_2 = 1$. If we think of outcome 1 as, say, Heads and outcome 2 as Tails, then we can rewrite this equivalently as $\alpha_1 = \alpha$, $\alpha_2 = 1 - \alpha$ and $x = x_1$ which is 1 when the outcome is Heads, and 0 when it is tails. Then $\alpha_1^{x_1} \alpha_2^{x_2} = \alpha^x (1-\alpha)^{1-x}$. Therefore, we can think of this distribution as the multidimensional generalization of the Bernoulli.

Other distributions we discussed for the univariate case can be extended to the multivariate case. The extension does not simply correspond to separately modelling each component of the vector, independently. Rather, the multivariate extensions typically allow us to reason about relationships between the variables. For the above multinomial, for example, we know if one variable has a successful outcome, and $n = 1$, then we know another cannot have a successful outcome. This is very different from separately modelling each component with a Bernoulli, which allows for multiple variables to be 1. For example, $(1, 1, 0, 0)$ might a possible outcome if each entry is modelled separately as a Bernoulli, but would not be possible under the multinomial with $n = 1$. Similarly, the extension of the Poisson to the multivariate Poisson considers relationships—like covariances—between the variables. We do not use the multivariate Poisson distribution in these notes, and so we do not discuss it further.

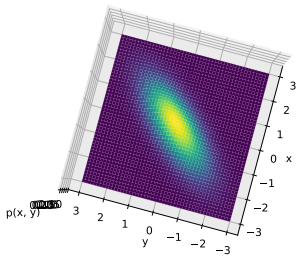
Exercise 3: The extension of the uniform distribution to the multivariate case is relatively straightforward, because each dimension is independent. Write down the pmf, assuming $\mathcal{X}_i = \{1, 2, \ldots, n\}$ for all i. Recall that for a univariate pmf $p(x) = 1/n$, with outcome space $x \in \{1, 2, \ldots, n\}$. Verify that your multidimensional uniform is a valid pmf. □



$$\mathbf{\Sigma} = \begin{bmatrix} 1.0 & 0 \\ 0 & 1.0 \end{bmatrix}$$



$$\mathbf{\Sigma} = \begin{bmatrix} 1.0 & 0.75 \\ 0.75 & 1.0 \end{bmatrix}$$



$$\mathbf{\Sigma} = \begin{bmatrix} 1.0 & 0.75 \\ 0.75 & 1.0 \end{bmatrix}$$

Figure 2.1: The multivariate Gaussian distribution with $d = 2$, for two different covariances. In (a), the covariance between random variables is zero, with only entries on the diagonal. In (b) and (c), the two components are correlated. This means that they have unit variance and a positive correlation, where if one is larger then it is more likely that the other is larger, and if one is smaller then it is more likely that the other is smaller. This correlation is emphasized in (c), looking from above, where we can clearly see the strong positive correlation between x and y: the density is higher for pairs where x and y are similar.

For the continuous case, any function $p : \mathcal{X} \to [0, \infty)$ is a probability density function (pdf) if

$$\int_{\mathcal{X}} p(\mathbf{x}) d\mathbf{x} = \int_{\mathcal{X}_1} \cdots \int_{\mathcal{X}_d} p(x_1, x_2, \ldots, x_d) dx_1 \cdots dx_d = 1.$$

For example, if $\mathcal{X}_i = \mathbb{R}$, giving $\mathcal{X} = \mathbb{R}^d$, this integral is

$$\int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} p(x_1, x_2, \ldots, x_d) dx_1 \cdots dx_d.$$

If $\mathcal{X}_i = [-1, 1]$, giving $\mathcal{X} = [-1, 1]^d$, this integral is

$$\int_{-1}^{1} \cdots \int_{-1}^{1} p(x_1, x_2, \ldots, x_d) dx_1 \cdots dx_d.$$

Recall that the density $p(\mathbf{x})$ at a point $\mathbf{x}$ can be greater than one, which is why the range of p is $[0, \infty)$.

The most useful multivariate generalization for us to consider is that for the Gaussian distribution. The *multivariate Gaussian distribution* is a generalization of the Gaussian or normal distribution to the d-dimensional case, with $\mathcal{X} = \mathbb{R}^d$. It is defined as

$$p(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^d |\mathbf{\Sigma}|}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\top \mathbf{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right), \tag{2.1}$$

with parameters $\boldsymbol{\mu} \in \mathbb{R}^d$ and a d-by-d covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}$. We will refer to this distribution as $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$; it is depicted in Figure 2.1.

This formula contains several new concepts, so let us step through it slowly. First, recall the definition of the covariance matrix. It is a two-dimensional array, where each entry corresponds to the covariance between two random variables

$$\begin{aligned}\Sigma_{ij} &= \text{Cov}[X_i, X_j] \\ &= \mathbb{E}\left[(X_i - \mathbb{E}\left[X_i\right])\left(X_j - \mathbb{E}\left[X_j\right]\right)\right]\end{aligned}$$

where our multivariate random variable is $\boldsymbol{X} = (X_1, X_2, \ldots, X_d)$. Notice that the diagonal of this matrix corresponds to the variances of each entry

$$\Sigma_{ii} = \text{Cov}[X_i, X_i] = \text{Var}[X_i]$$

The formula includes $|\boldsymbol{\Sigma}|$, which is called the *determinant* of $\boldsymbol{\Sigma}$. For us, this determinant reflects the overall variance: if it is large in most directions, this determinant will be large, and if there are some directions where the density is very peaked (low variance), then it could be very small. In this simplest case, when the covariance between random variables is zero, the covariance is a diagonal matrix. Namely, it has the variances on the main diagonal, with zeros everywhere else

$$\boldsymbol{\Sigma} = \begin{bmatrix} \text{Var}[X_1] & 0 & \ldots & 0 & 0 \\ 0 & \text{Var}[X_2] & 0 & \ldots & 0 \\ & \ldots & & & \\ 0 & 0 & \ldots & 0 & \text{Var}[X_d] \end{bmatrix} = \begin{bmatrix} \sigma_1^2 & 0 & \ldots & 0 & 0 \\ 0 & \sigma_2^2 & 0 & \ldots & 0 \\ & \ldots & & & \\ 0 & 0 & \ldots & 0 & \sigma_d^2 \end{bmatrix}$$

where we write $\sigma_i^2 \stackrel{\text{def}}{=} \text{Var}[X_i]$, in other words where σ_i is the standard deviation for the ith variable. Then the determinant is $|\boldsymbol{\Sigma}| = \sigma_1^2 \sigma_2^2 \ldots \sigma_d^2$, the product of these variances. Therefore, the first term includes a normalization with the magnitudes of the variances.

Let us continue with this simpler case, to understand the second component with a matrix inverse. In Section 1.5, we discussed that the inverse of a diagonal matrix is the inverse of each scalar on the diagonal

$$\boldsymbol{\Sigma}^{-1} = \begin{bmatrix} 1/\sigma_1^2 & 0 & \ldots & 0 & 0 \\ 0 & 1/\sigma_2^2 & 0 & \ldots & 0 \\ & \ldots & & & \\ 0 & 0 & \ldots & 0 & 1/\sigma_d^2 \end{bmatrix}$$

When we take the matrix-vector product $\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})$, we are scaling each component by the correspond diagonal element

$$\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}) = \begin{bmatrix} \frac{1}{\sigma_1^2}(x_1 - \mu_1) \\ \frac{1}{\sigma_2^2}(x_2 - \mu_2) \\ \ldots \\ \frac{1}{\sigma_d^2}(x_d - \mu_d) \end{bmatrix} \tag{2.2}$$

Let $\mathbf{v} = \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})$. Then, we take the dot product again with $(\mathbf{x} - \boldsymbol{\mu})$ to get

$$(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}) = (\mathbf{x} - \boldsymbol{\mu})^\top \mathbf{v} = \sum_{j=1}^{d}(x_j - \mu_j)v_j$$
$$= \sum_{j=1}^{d}(x_j - \mu_j)\tfrac{1}{\sigma_j^2}(x_j - \mu_j) = \sum_{j=1}^{d}\tfrac{1}{\sigma_j^2}(x_j - \mu_j)^2.$$

Therefore, this term corresponds to a weighted sum of squared differences to the mean.

In fact, recall that the squared ℓ_2 norm is $\|\mathbf{v}\|_2^2 = \sum_{j=1}^{d} v_j^2$. We can see that $\|\mathbf{x} - \boldsymbol{\mu}\|_2^2 = \sum_{j=1}^{d}(x_j - \mu_j)^2$. Therefore, this term corresponds to an ℓ_2 norm, but weighted by the magnitude of the variance in each dimension. If the variance is small in one direction, this amplifies the difference; if it is large, it downweights it. This makes sense for a Gaussian. A big variance implies that outcomes further from the mean still have reasonably high density. A small variance means that the Gaussian is peaked, and moving even a small amount away from the mean significantly decreases the density.

Exercise 4: Use the formula for matrix-vector multiplication to show the step for the equality in Equation (2.2). □

Example 1: Let us look at an example with 2-dimensions, $d = 2$. Let

$$\boldsymbol{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0.3 \end{bmatrix} \qquad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix} = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix}$$

This Gaussian has no covariance between the two variables. The first variable has much higher variance (10) than the second, which has a variance of 1. This means the Gaussian is wider in the first dimension—and flatter—and narrower and more peaked in the second dimension. Further, it is shifted in the first dimension x_1 to be more negative—centered around $\mu_1 = -1$—and shifted to be slightly positive in the second dimension x_2. □

Now let us consider what the Gaussian looks like with covariance between the variables, namely non-zeros on the off-diagonal. This causes the distribution to become more skewed, as shown in Figure 2.1. Further, the determinant and inverse become a bit more involved to compute, since $\boldsymbol{\Sigma}$ is no longer a diagonal matrix.

The covariance matrix is symmetric and positive definite; i.e., $\boldsymbol{\Sigma} \succ 0$. This means that the eigenvalues are positive. Recall that the eigenvalue decomposition for a symmetric matrix is $\boldsymbol{\Sigma} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top$ for orthonormal matrix $\mathbf{U} \in \mathbb{R}^{d \times d}$ and diagonal matrix $\boldsymbol{\Lambda} \in \mathbb{R}^{d \times d}$. Every symmetric matrix has an eigenvalue decomposition, and so $\boldsymbol{\Sigma}$ has an eigenvalue decomposition. Now the determinant is

$$|\boldsymbol{\Sigma}| = \Pi_{i=1}^{d}\lambda_j \tag{2.3}$$

where the $\lambda_j \in \mathbb{R}$ are the eigenvalues on the diagonal of $\boldsymbol{\Lambda}$. The determinant reflects the volume spanned by the matrix. If the eigenvalues are large, then a larger volume is occupied. You can think of the eigenvalues $\boldsymbol{\Lambda}$ as the diagonal variances in the rotated space, where rotation does not affect the width or variability in each (rotated) dimension.

2.2 Properties of Expectations

Expectations for multivariate variables are the same as in the univariate case.

$$\mathbb{E}[\boldsymbol{X}] \stackrel{\text{def}}{=} \begin{cases} \sum_{\mathbf{x}\in\mathcal{X}} \mathbf{x} p(\mathbf{x}) & \text{if } \boldsymbol{X} \text{ is discrete} \\ \\ \int_{\mathcal{X}} \mathbf{x} p(\mathbf{x}) d\mathbf{x} & \text{if } \boldsymbol{X} \text{ is continuous} \end{cases} \tag{2.4}$$

Notice further that this expectation is actually element-wise

$$\mathbb{E}[\boldsymbol{X}] = \begin{bmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \dots \\ \mathbb{E}[X_d] \end{bmatrix} \tag{2.5}$$

where each expectation $\mathbb{E}[X_j]$ uses the marginal $p(x_j)$. Recall that a marginal is the distribution over a single variable, computed from the larger joint distribution. For example, if all the variables are discrete, for a given $x_j \in \mathcal{X}_j$,

$$p(x_j) = \sum_{x_1\in\mathcal{X}_1} \dots \sum_{x_{j-1}\in\mathcal{X}_{j-1}} \sum_{x_{j+1}\in\mathcal{X}_{j+1}} \dots \sum_{x_d\in\mathcal{X}_d} p(x_1, x_2, \dots, x_{j-1}, x_j, x_{j+1}, \dots, x_d)$$

This means it is straightforward to define the expectation, even for $\boldsymbol{X}$ composed of both discrete and continuous variables. For example, if X_1 is discrete and X_2 is continuous, then for $\boldsymbol{X} = [X_1, X_2]$, the expectation for the first element uses sums and for the second element uses integrals.

The marginals themselves can also be mixed. Imagine we have three variables, where X_2 is discrete and X_3 is continuous. Then, regardless of the type for X_1, we can get the marginal

$$p(x_1) = \sum_{x_2\in\mathcal{X}_2} \int_{\mathcal{X}_3} p(x_1, x_2, x_3) dx_3 = \int_{\mathcal{X}_3} \left(\sum_{x_2\in\mathcal{X}_2} p(x_1, x_2, x_3) \right) dx_3$$

where the order of the sum and integral is not relevant. The definition of the marginal only relies on the types for the variables we marginalize over, namely X_2 and X_3. The resulting marginal is a pmf if X_1 is discrete and is a pdf if X_1 is continuous.

Exercise 5: Show that we get the elementwise expectation in Equation (2.5), using the definition of expectation in Equation (2.4). □

Here we review, without proofs, some useful properties of expectations. We can generically consider multivariate random variables, $\boldsymbol{X} \in \mathbb{R}^d$ and $\boldsymbol{Y} \in \mathbb{R}^d$, for $d \in \mathbb{N}$, with univariate random variables as a special case (for $d = 1$). For a constant $c \in \mathbb{R}$, it holds that:

1. $\mathbb{E}[c\boldsymbol{X}] = c\mathbb{E}[\boldsymbol{X}]$
2. $\mathbb{E}[\boldsymbol{X} + \boldsymbol{Y}] = \mathbb{E}[\boldsymbol{X}] + \mathbb{E}[\boldsymbol{Y}]$
3. $\text{Var}[c] = 0$ ▷ the variance of a constant is zero

4. $\text{Var}[\boldsymbol{X}] \succeq 0$ (i.e., is positive semi-definite), where for $d = 1$, $\text{Var}[\boldsymbol{X}] \geq 0$ is a scalar. Note that $\text{Var}[\boldsymbol{X}]$ is shorthand for $\text{Cov}[\boldsymbol{X}, \boldsymbol{X}]$.
5. $\text{Var}[c\boldsymbol{X}] = c^2\text{Var}[\boldsymbol{X}]$.
6. $\text{Cov}[\boldsymbol{X}, \boldsymbol{Y}] = \mathbb{E}[(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])(\boldsymbol{Y} - \mathbb{E}[\boldsymbol{Y}])^\top] = \mathbb{E}[\boldsymbol{X}\boldsymbol{Y}^\top] - \mathbb{E}[\boldsymbol{X}]\mathbb{E}[\boldsymbol{Y}]^\top$
7. $\text{Var}[\boldsymbol{X} + \boldsymbol{Y}] = \text{Var}[\boldsymbol{X}] + \text{Var}[\boldsymbol{Y}] + 2\text{Cov}[\boldsymbol{X}, \boldsymbol{Y}]$
8. $\text{Var}[\boldsymbol{X}_1 + \boldsymbol{X}_2 + \ldots + \boldsymbol{X}_m] = \sum_{i=1}^{m}\sum_{j=1}^{m}\text{Cov}[\boldsymbol{X}_i, \boldsymbol{X}_j] = \sum_{i=1}^{m}\text{Var}[\boldsymbol{X}_i] + 2\sum_{1\leq i<j\leq m}\text{Cov}[\boldsymbol{X}_i, \boldsymbol{X}_j]$

In addition, if $\boldsymbol{X}$ and $\boldsymbol{Y}$ are independent random variables of the same dimension, it holds that:

1. $\mathbb{E}[X_i Y_j] = \mathbb{E}[X_i]\,\mathbb{E}[Y_j]$ for all i, j
2. $\text{Var}[\boldsymbol{X} + \boldsymbol{Y}] = \text{Var}[\boldsymbol{X}] + \text{Var}[\boldsymbol{Y}]$
3. $\text{Cov}[\boldsymbol{X}, \boldsymbol{Y}] = 0$.

2.3 Mixtures of Distributions

In previous sections we saw that random variables are often described using particular families of probability distributions. This approach can be generalized by considering mixtures of distributions: convex combinations of other probability distributions. This allows us to take relatively simply distributions, like Gaussians, and produce much more complex distributions. An example is given in Figure 2.2, where even with just two Gaussians, we can already see that we can model many more densities.

Formally, a *mixture model* $p(\mathbf{x})$ is defined on a set of m probability distributions, $\{p_i(\mathbf{x})\}_{i=1}^m$

$$p(\mathbf{x}) = \sum_{i=1}^{m} w_i p_i(\mathbf{x}), \tag{2.6}$$

where $\boldsymbol{w} = (w_1, w_2, \ldots, w_m)$ is a set of non-negative real numbers such that $\sum_{i=1}^m w_i = 1$. We refer to $\boldsymbol{w}$ as mixing coefficients. A linear combination with such coefficients is called a convex combination.[1] For discrete random variables $\boldsymbol{X}$, the $p_i : \mathcal{X} \to [0, 1]$ must be pmfs, and the resulting $p : \mathcal{X} \to [0, 1]$ is also a pmf. For continuous random variables $\boldsymbol{X}$, the $p_i : \mathcal{X} \to [0, \infty)$ must be pdfs, and the resulting $p : \mathcal{X} \to [0, \infty)$ is also a pdf.

Exercise 6: Assume $\boldsymbol{X}$ is discrete and verify that p satisfies the rules for pmfs, namely that $p(\mathbf{x}) \geq 0$ for all $\mathbf{x}$ and $\sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x}) = 1$. Use the fact that we know each p_i is a valid pmf, and the constraints on the coefficients w_i. □

[1] Whenever we use non-negative weights that sum to 1, we are interpolating between the set of items. This interpolation results in a *convex set*, which is why this is called a convex combination. A set C is convex if for any two $x, y \in C$ we have $\lambda x + (1 - \lambda)y \in C$ for any $\lambda \in [0, 1]$.

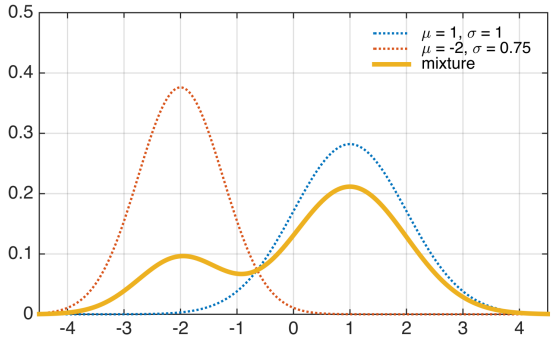


Figure 2.2: A Gaussian Mixture Model, where the two Gaussians are mixed with coefficients $w_1 = 0.25$ *on the red Gaussian (leftmost) and* $w_2 = 0.75$ *on the blue Gaussian (rightmost). Namely, if we label the red Gaussian as* p_1 *and the blue as* p_2*, then the yellow line is* $p(x) = w_1 p_1(x) + w_2 p_2(x) = 0.25\mathcal{N}(-2, 0.75) + 0.75\mathcal{N}(1, 1)$.

Here we will briefly look into expectations for mixture distributions. Let $\boldsymbol{X}_i$ be the (implicit) random variable described by the distribution p_i. Note it is implicit, since we are not modeling $\boldsymbol{X}_i$; rather we are modeling $\boldsymbol{X}$. For any function f on $\boldsymbol{X}$,

$$\begin{aligned}
\mathbb{E}\left[f(\boldsymbol{X})\right] &= \int_{\mathcal{X}} f(\mathbf{x}) p(\mathbf{x}) dx \\
&= \int_{\mathcal{X}} f(\mathbf{x}) \sum_{i=1}^{m} w_i p_i(\mathbf{x}) dx \\
&= \sum_{i=1}^{m} w_i \int_{\mathcal{X}} f(\mathbf{x}) p_i(\mathbf{x}) dx \\
&= \sum_{i=1}^{m} w_i \mathbb{E}[f(\boldsymbol{X}_i)].
\end{aligned}$$

We get the same outcome for discrete $\boldsymbol{X}$.

Consider the simpler univariate setting, where $d = 1$. We can apply this formula to obtain the mean, when $f(x) = x$ and the variance, when $f(x) = (x - E[X])^2$, of the random variable X as

$$\mathbb{E}[X] = \sum_{i=1}^{m} w_i \mathbb{E}[X_i],$$

and

$$\mathrm{Var}[X] = \sum_{i=1}^{m} w_i \mathrm{Var}[X_i] + \sum_{i=1}^{m} w_i \left(\mathbb{E}[X_i] - \mathbb{E}[X]\right)^2.$$

The variance corresponds to the weighted sum of the variances of the individual mixture components, as well as the weighted sum of the distances between the means of the components and the mean of the mixture. Intuitively, this makes sense. Consider when we mix two Gaussians, as in Figure 2.2. If the means of the two Gaussians are far apart, then the

mean of the mixture likely lies somewhere between them and these distances in the second term will be quite large. The samples x from this mixture can vary widely, either centered around the mode of the first Gaussian or much further away at the mode of the second Gaussian.

Notice that it is very different to take a convex combination of random variables versus a convex combination of their distributions. Mixture models use convex combinations of distributions. For the above example, consider if we instead use $Y = \sum_{i=1}^{m} w_i X_i$. Then the expectation of this Y is actually the same as the mixture X, but the variances are different

$$\text{Var}[Y] = \text{Var}\left[\sum_{i=1}^{m} w_i X_i\right] = \sum_{i=1}^{m} w_i \text{Var}[X_i] + \sum_{i=1}^{m}\sum_{j=1}^{m} \text{Cov}[X_i, X_j]$$

For example, if the X_i are independent, then the covariance terms disappear and the variance is simply $\sum_{i=1}^{m} w_i \text{Var}[X_i]$. In general, the distributions can be very different. For example, if Y is the convex combination of two Gaussian random variables X_i, then Y is itself again Gaussian (a unimodal density). The mixture model of these two Gaussians, however, has the bimodal structure we see in Figure 2.2.

Example 2: We use a signal communications example to give another example that the distribution for a convex combination of two random variables X and Y does not correspond to a combination of their distributions. Consider transmission of a single binary digital signal (bit) over a noisy communication channel. The magnitude of the signal X emitted by the source is equally likely to be 0 or 1 Volt. The signal is sent over a transmission line (e.g., radio communication, optical fiber, magnetic tape) in which a Gaussian noise component Y is added to X, and receive $Z = X + Y$.

We can model this as follows. We have X : Bernoulli(α) and Y : Gaussian(μ, σ^2). It can be shown that

$$p(z) = \alpha \cdot \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(z-\mu-1)^2} + (1-\alpha) \cdot \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(z-\mu)^2}.$$

(We omit the steps to obtain this, because that is not the point of this example.) We see that $p(z)$ is a mixture of two normal distributions $\mathcal{N}(\mu+1, \sigma^2)$ and $\mathcal{N}(\mu, \sigma^2)$ with coefficients $w_1 = \alpha$ and $w_2 = 1 - \alpha$. But the underlying variables producing Z involve the sum of a Gaussian and a Bernoulli random variable. □

2.4 Revisiting MLE with Multivariate Gaussians

A good way to refresh your maximum likelihood estimation (MLE) knowledge is to apply it to a new distribution: multivariate Gaussians. Assume you would like to find parameters $\boldsymbol{\mu}, \boldsymbol{\Sigma}$ for a multivariate Gaussian $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ where $\boldsymbol{\Sigma}$ is positive definite and symmetric. (We write $\boldsymbol{\Sigma} \succ 0$ to indicate that this matrix is positive definite.) Recall that the pdf, introduced in Equation 2.1, is

$$p(\mathbf{x}) = (2\pi)^{-d/2} |\boldsymbol{\Sigma}|^{-1/2} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right).$$

The negative log likelihood for a given $\mathbf{x}$ for a Gaussian is

$$\begin{aligned} -\ln p(\mathbf{x}) &= -\ln(2\pi)^{-d/2} - \ln|\boldsymbol{\Sigma}|^{-1/2} - \ln\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \\ &= -\ln(2\pi)^{-d/2} + \tfrac{1}{2}\ln|\boldsymbol{\Sigma}| + \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \end{aligned}$$

Now we can compute the gradient with respect to both of our parameters $\boldsymbol{\mu}, \boldsymbol{\Sigma}$, for the full objective and find a stationary point. The objective for $\boldsymbol{\theta} = (\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is

$$c(\boldsymbol{\theta}) = \sum_{i=1}^n c_i(\boldsymbol{\theta}) \quad \text{where } c_i(\boldsymbol{\theta}) \stackrel{\text{def}}{=} -\ln p(\mathbf{x}_i) = -\ln(2\pi)^{-d/2} + \tfrac{1}{2}\ln|\boldsymbol{\Sigma}| + \frac{1}{2}(\mathbf{x}_i-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i-\boldsymbol{\mu})$$

Notice that we can still consider our set of parameters to be a vector $\boldsymbol{\theta} \in \mathbb{R}^{d+d^2}$, where the d elements correspond to $\boldsymbol{\mu}$ and $\boldsymbol{\theta}_{d+1} = \Sigma_{11}$, $\boldsymbol{\theta}_{d+2} = \Sigma_{21}$, $\ldots$, $\boldsymbol{\theta}_{2d} = \Sigma_{d1}$, $\boldsymbol{\theta}_{2d+1} = \Sigma_{12}$, $\ldots$, $\boldsymbol{\theta}_{d+d^2} = \Sigma_{dd}$. When we compute the stationary point, we are computing partial derivatives for each $\boldsymbol{\theta}_j$ and setting them to zero. This corresponds to computing partial derivatives for each element of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ and finding the point where those partial derivatives are zero. We can use gradient descent, but in this case we can actually obtain a closed form solution by solving for $\boldsymbol{\mu}, \boldsymbol{\Sigma}$ such that both $\nabla_{\boldsymbol{\mu}} c(\boldsymbol{\theta}) = 0$ and $\nabla_{\boldsymbol{\Sigma}} c(\boldsymbol{\theta}) = 0$.

To get these partial derivatives, we will compute them for each c_i.

$$\begin{aligned} \nabla_{\boldsymbol{\mu}} -\ln p(\mathbf{x}) &= \nabla_{\boldsymbol{\mu}} -\ln(2\pi)^{-d/2} + \nabla_{\boldsymbol{\mu}} \tfrac{1}{2}\ln|\boldsymbol{\Sigma}| + \nabla_{\boldsymbol{\mu}} \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \\ &= \mathbf{0} + \mathbf{0} + \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}-\mathbf{x}) \end{aligned}$$

where the last follows from the fact that $\nabla_{\boldsymbol{\mu}} \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}-\mathbf{x})$. You can check this as an exercise, either using the rules for derivatives with vectors, discussed in Section 1.5.3, or by computing the partial derivatives for each μ_j. We can therefore see that, for any full rank (invertible) $\boldsymbol{\Sigma}$, we have

$$\begin{aligned} &\nabla_{\boldsymbol{\mu}} -\sum_{i=1}^n \ln p(\mathbf{x}_i) = \boldsymbol{\Sigma}^{-1}\sum_{i=1}^n(\boldsymbol{\mu}-\mathbf{x}_i) = \mathbf{0} \\ &\implies \sum_{i=1}^n(\boldsymbol{\mu}-\mathbf{x}_i) = \mathbf{0} \qquad \triangleright \text{multiply both sides by } \boldsymbol{\Sigma} \implies n\boldsymbol{\mu} = \sum_{i=1}^n \mathbf{x}_i \\ &\implies \boldsymbol{\mu}^* = \tfrac{1}{n}\sum_{i=1}^n \mathbf{x}_i \end{aligned}$$

We know that this stationary point is a global minimum because the objective is convex: it is a weighted quadratic objective, with positive definite weighting $\boldsymbol{\Sigma}^{-1}$.

Similarly, we can compute the gradient w.r.t. $\boldsymbol{\Sigma}$ and solve for a stationary point. This answer is again intuitive: it is the sample covariance. We leave this as an exercise.

Exercise 7: Show the stationary point w.r.t. $\boldsymbol{\Sigma}$ is the sample covariance $\frac{1}{n}\sum_{i=1}^n \mathbf{x}_i\mathbf{x}_i^\top$. □

This stationary point satisfies the conditions on $\boldsymbol{\Sigma}$, namely that it is symmetric and positive definite, if $n > d$. This is because an outer product $\mathbf{a}_i\mathbf{a}_i^\top$ is always symmetric, and the sum of symmetric matrices is symmetric. Further, the sum of outer products $\sum_{i=1}^n \mathbf{a}_i\mathbf{a}_i^\top$ has at most rank n. Therefore, if $n < d$, the rank will be less than d and $\boldsymbol{\Sigma}$ will not be full rank.

Because we usually assume $n > d$, we will usually get a full rank $\mathbf{\Sigma}$ and we can conclude that we have a global minimum.

Exercise 8: Typically, once $n > d$, we expect $\mathbf{\Sigma}$ to be full rank and so positive definite. However it is possible that $n > d$ and we get a $\mathbf{\Sigma}$ that is not full rank. When might this happen and why? Hint: start by considering a case where you have two repeated samples $\mathbf{x}_3 = \mathbf{x}_5$ and $n = d$. □

Exercise 9: The above is the MLE solution. We could instead consider a MAP solution, where we incorporate a prior on our variables. What prior information might we want to incorporate on $\mathbf{\Sigma}$? □

2.5 Entropy and KL Divergence

We have so far mainly discussed statistics on distributions. Another set of metrics reflects their information content, which can be seen as the level of stochasticity or randomness in the random variables. The *entropy* of a random variable is defined as

$$H(X) = \begin{cases} -\sum_{x \in \mathcal{X}} p(x) \log p(x) & \text{if } X \text{ is discrete} \\ -\int_{\mathcal{X}} p(x) \log p(x) dx & \text{if } X \text{ is continuous} \end{cases}$$

For continuous RVs it is called the *differential entropy*. The uniform distribution has the highest entropy, and a perfectly peaked (deterministic) distribution has the lowest entropy. The entropy, for us, is primarily useful to define the Kullback-Leibler (KL) divergence.

The KL divergence between probability distributions $p(x)$ and $q(x)$ on $\mathcal{X} = \mathbb{R}$ is

$$D_{\mathrm{KL}}(p||q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx.$$

In information theory, KL divergence has a natural interpretation of the inefficiency of signal compression when the code is constructed using a suboptimal distribution $q(x)$ instead of the correct (but unknown) distribution $p(x)$ according to which the data has been generated. However, the KL divergence is often used as a measure of divergence between two probability distributions. Although this divergence is not a metric (it is not symmetric and does not satisfy the triangle inequality) it has important theoretical properties in that (i) it is always non-negative and (ii) it is equal to zero if and only if $p(x) = q(x)$.

One interesting note is the relationship between maximum likelihood estimation and KL divergence. Let $p(x|\theta)$ be the estimated probability distribution and $p(x|\theta_0)$ the underlying (true) distribution according to which the data set $\mathcal{D} = \{x_i\}_{i=1}^n$ was generated. The KL divergence between $p = p(\cdot|\theta_0)$ and $q = p(\cdot|\theta)$ is[2]

$$\begin{aligned} D_{\mathrm{KL}}(p(\cdot|\theta_0)||p(\cdot|\theta)) &= \int_{\mathcal{X}} p(x|\theta_0) \log \frac{p(x|\theta_0)}{p(x|\theta)} dx \\ &= \int_{\mathcal{X}} p(x|\theta_0) \log \frac{1}{p(x|\theta)} dx + \int_{\mathcal{X}} p(x|\theta_0) \log p(x|\theta_0) dx. \end{aligned}$$

[2]Note that if functions have two variables $f(x, y)$, when we want to talk about the function on one variable, with the other variable y fixed, then we write function $g = f(\cdot, y)$. Here, we write $p(\cdot|\theta)$ to indicate that we have the density over x for a given θ. If we wrote $p(x|\theta)$, that is the density for a specific x. Our goal is to specify the KL between the two densities, across all x, so we write $p(\cdot|\theta_0)$ and $p(\cdot|\theta)$.

$$= \int_{\mathcal{X}} p(x|\theta_0) \log \frac{1}{p(x|\theta)} dx - (- \int_{\mathcal{X}} p(x|\theta_0) \log p(x|\theta_0) dx.)$$
$$= \int_{\mathcal{X}} p(x|\theta_0) \log \frac{1}{p(x|\theta)} dx - \int_{\mathcal{X}} p(x|\theta_0) \log \frac{1}{p(x|\theta_0)} dx.$$

The second term in the above equation is the (differential) entropy of the true distribution and is not influenced by our choice of the model θ. The first term, on the other hand, can be expressed as

$$\int_{\mathcal{X}} p(x|\theta_0) \log \frac{1}{p(x|\theta)} dx = - \int_{\mathcal{X}} p(x|\theta_0) \log p(x|\theta) dx = -\mathbb{E}[\log p(X|\theta)].$$

where the expectation is taken with respect to $p(x|\theta_0)$. To see why, define $f(x) \stackrel{\text{def}}{=} \log p(x|\theta)$. Then $\mathbb{E}[f(X)] = \int_{\mathcal{X}} p(x|\theta_0) f(x) dx = \int_{\mathcal{X}} p(x|\theta_0) \log p(x|\theta) dx$.

Therefore, minimizing the expected negative log-likelihood $-\mathbb{E}[\log p(X|\theta)]$ minimizes the KL divergence between $p(x|\theta)$ and $p(x|\theta_0)$:

$$\operatorname*{argmin}_{\theta} D_{\mathrm{KL}}(p(\cdot|\theta_0)||p(\cdot|\theta)) = \operatorname*{argmin}_{\theta} -\mathbb{E}[\log p(X|\theta)].$$

Using the strong law of large numbers, we know

$$\frac{1}{n} \sum_{i=1}^{n} \log p(x_i|\theta) \quad \stackrel{a.s.}{\to} \quad \mathbb{E}[\log p(X|\theta)]$$

when $n \to \infty$. Thus, when the data set is sufficiently large, minimizing the negative log likelihood is very nearly like minimizing this KL divergence.

Exercise 10: The KL divergence is not symmetric: using $D_{\mathrm{KL}}(p(\cdot|\theta_0)||p(\cdot|\theta))$ is different from $D_{\mathrm{KL}}(p(\cdot|\theta)||p(\cdot|\theta_0))$. The key difference is which distribution is used to compute the expected value. Imagine we instead tried to use $D_{\mathrm{KL}}(p(\cdot|\theta)||p(\cdot|\theta_0))$ as our objective. What issues arise? □

Chapter 3

Revisiting Linear Regression

Given a data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ the objective is to learn the relationship between features and the target. We also wrote this as data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\mathbf{y} \in \mathbb{R}^n$, where $\mathbf{X}(i,:) = \mathbf{x}_i$ and $\mathbf{y}(i) = y_i$. In linear regression, we assumed the function f is a linear function

$$f(\mathbf{x}) = \sum_{j=1}^{d} w_j x_j = \langle \mathbf{x}, \mathbf{w} \rangle = \mathbf{x}\mathbf{w}$$

where we assume $x_1 = 1$ so that w_0 corresponds to an intercept term. Note also that we assume that $\mathbf{x} \in \mathbb{R}^{1 \times d}$ is a row vector, since it is a row of the data matrix $\mathbf{X}$, and so $\mathbf{x}\mathbf{w}$ corresponds to the dot product between these two vectors. Throughout these notes, we will treat a row of $\mathbf{X}$, that is $\mathbf{x}$, as a row vector. We derived updates to obtain the maximum likelihood solution—the ordinary least squares solution—and the ℓ_2 regularized solution, sometimes called the ridge regression solution. The SGD update for linear regression is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \left(\mathbf{x}_i \mathbf{w}_t - y_i\right) \mathbf{x}_i^\top$$

For ℓ_2 regularization, with regularization parameter $\lambda > 0$, we have update

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \left[\left(\mathbf{x}_i \mathbf{w}_t - y_i\right) \mathbf{x}_i^\top + \lambda \mathbf{w}_t\right]$$

In this chapter, we revisit the properties of this solution. We first look at the closed form solution for OLS and ridge regression, and then analyze the stability of this solution.

3.1 Ordinary Least-Squares (OLS) Regression

Recall that when we wrote down the maximum likelihood formulation for linear regression, it corresponded to solving the following minimization problem

$$\mathbf{w}_{\text{MLE}} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^{n} (\mathbf{x}_i \mathbf{w} - y_i)^2.$$

We found that the gradient of this objective was

$$\nabla \sum_{i=1}^{n} (\mathbf{x}_i \mathbf{w} - y_i)^2 = 2 \left(\sum_{i=1}^{n} \mathbf{x}_i^\top \mathbf{x}_i \right) \mathbf{w} - 2 \sum_{i=1}^{n} \mathbf{x}_i^\top y_i$$

where $\mathbf{x}_i^\top \mathbf{x}_i \in \mathbb{R}^{d \times d}$ is an outer product because $\mathbf{x}$ is a row vector. We use this gradient to solve for $\nabla \sum_{i=1}^n (\mathbf{x}_i \mathbf{w} - y_i)^2 = \mathbf{0}$ to get

$$\mathbf{w} = \left(\sum_{i=1}^{n} \mathbf{x}_i^\top \mathbf{x}_i \right)^{-1} \sum_{i=1}^{n} \mathbf{x}_i^\top y_i$$

Let us now rewrite this a bit more compactly using matrices. Notice first that when we multiply matrices, we are essentially summing over the inner dimension, with outer products for rows and columns of the matrices: for $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{n \times p}$,

$$\mathbf{AB} = \sum_{i=1}^{n} \mathbf{A}(:, i)\mathbf{B}(i, :)$$

where $\mathbf{A}(:, i)$ is the ith column of $\mathbf{A}$ and $\mathbf{B}(i, :)$ is the ith row of $\mathbf{B}$. We can see this by noticing that if we index into a specific element $\mathbf{C}(j, k)$ for $\mathbf{C} = \mathbf{AB}$, then it is the dot product between the jth row in $\mathbf{A}$ and kth column in $\mathbf{B}$:

$$\mathbf{C}(j, k) = \mathbf{A}(j, :)\mathbf{B}(:, k) = \sum_{i=1}^{n} \mathbf{A}(j, i)\mathbf{B}(i, k).$$

We can similarly rewrite the sum of the outer products of $\mathbf{x}_i$ above using

$$\mathbf{X}^\top \mathbf{X} = \sum_{i=1}^{n} \mathbf{X}(i, :)^\top \mathbf{X}(i, :) = \sum_{i=1}^{n} \mathbf{x}_i^\top \mathbf{x}_i.$$

Additionally, we can write $\sum_{i=1}^{n} \mathbf{x}_i^\top y_i = \mathbf{X}^\top \mathbf{y}$ where again the dot product plays the role of the sum. Then we can write the OLS solution as

$$\mathbf{w} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

We could also have started by writing the objective using these matrices and vectors

$$\mathbf{w}_{\text{MLE}} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \|\mathbf{Xw} - \mathbf{y}\|_2^2$$

where the ℓ_2 norm is defined as $\|\mathbf{a}\|_2^2 = \mathbf{a}^\top \mathbf{a} = \sum_{i=1}^{n} a_i^2$. Notice that $\hat{\mathbf{y}} \stackrel{\text{def}}{=} \mathbf{Xw}$ is the vector of predictions, where we dot product $\mathbf{w}$ with each row of $\mathbf{X}$ (each sample), giving each element of the vector $\hat{\mathbf{y}}$ as $\hat{y}_i = \mathbf{x}_i \mathbf{w}$. Given this form, we could use the simple rules for computing gradients for vectors, in Table 1.1.

Exercise 11: Use the rules from Table 1.1 to compute the gradient of $\|\mathbf{Xw} - \mathbf{y}\|_2^2$ wrt $\mathbf{w}$. □

In either case, when we set the derivative to zero, and rearrange, we get

$$\mathbf{X}^\top \mathbf{Xw} = \mathbf{X}^\top \mathbf{y} \implies \mathbf{w}_{\text{MLE}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \tag{3.1}$$

We can see that the predictions on the training set from this OLS solution are

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}_{\text{MLE}} = \mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

The matrix $\mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ is called the *projection matrix*, because it projects $\mathbf{y}$ to the column space of $\mathbf{X}$. In machine learning, we might say that it projects y to the space representable by a linear combination of our features: $\hat{\mathbf{y}}$ is the best linear approximation to $\mathbf{y}$ for the given features.

Example 3: Consider the data set $\mathcal{D} = \{(1, 1.2), (2, 2.3), (3, 2.3), (4, 3.3)\}$. We want to find the optimal coefficients of the least-squares fit for $f(x) = w_0 + w_1 x$ and then calculate the sum of squared errors on $\mathcal{D}$ after the fit. The OLS solution can be obtained using

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix}, \ \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, \ \mathbf{y} = \begin{bmatrix} 1.2 \\ 2.3 \\ 2.3 \\ 3.3 \end{bmatrix},$$

where a column of ones was added to $\mathbf{x}$ to allow for a non-zero intercept. Substituting $\mathbf{X}$ and $\mathbf{y}$ into Eq. (3.1) results in $\mathbf{w} = (0.7, 0.63)$ and the sum of square errors is 0.223. This solution is obtained by using a numerical library, which involves computing the matrix products and inverses. □

3.1.1 Extension to a Weighted Error Function

In some applications it is useful to consider minimizing the weighted error function

$$c(\mathbf{w}) = \sum_{i=1}^{n} b_i \left(\mathbf{x}_i \mathbf{w} - y_i\right)^2,$$

where $b_i > 0$ is a weighting for data point i. Expressing this in a matrix form, the goal is to minimize $(\mathbf{X}\mathbf{w} - \mathbf{y})^\top \mathbf{B} (\mathbf{X}\mathbf{w} - \mathbf{y})$, where $\mathbf{B} = \text{diag}(b_1, b_2, \ldots, b_n)$. Using a similar approach as above, it can be shown that the weighted least-squares solution $\mathbf{w}_\text{b}$ can be expressed as

$$\mathbf{w}_\text{b} = \left(\mathbf{X}^\top \mathbf{B} \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{B} \mathbf{y}.$$

In addition, it can be derived that

$$\mathbf{w}_\text{b} = \mathbf{w}_\text{MLE} + \left(\mathbf{X}^\top \mathbf{B} \mathbf{X}\right)^{-1} \mathbf{X}^\top (\mathbf{I} - \mathbf{B}) (\mathbf{X}\mathbf{w}_\text{MLE} - \mathbf{y}),$$

where $\mathbf{w}_\text{MLE}$ is provided by Eq. (3.1). We can see the solutions are identical when $\mathbf{B} = \mathbf{I}$, but also when $\mathbf{X}\mathbf{w}_\text{MLE} = \mathbf{y}$. In other words, if we can perfectly fit to $\mathbf{y}$, then the weighting does not influence the solution. This makes sense, since the weighting is trading off the error for different samples. If there is zero error for every sample, then there is no trade-off.

Exercise 12: Derive the above weighted solution, similarly to how we derived the closed form solution for linear regression. □

Exercise 13: We assumed $b_i > 0$. What if we set $b_i = 0$ for one sample i? How does it change the solution? What about if we set $b_i = -1$? □

3.1.2 Predicting Multiple Outputs Simultaneously

The extension to multiple outputs is straightforward, where now the target is an m-dimensional vector, $\mathbf{y} \in \mathbb{R}^m$, rather than a scalar, giving target matrix $\mathbf{Y} \in \mathbb{R}^{n \times m}$. Correspondingly, the weights are also a matrix $\mathbf{W} \in \mathbb{R}^{d \times m}$, giving prediction $\mathbf{x}^\top \mathbf{W} \in \mathbb{R}^m$, with error

$$c(\mathbf{W}) = \|\mathbf{X}\mathbf{W} - \mathbf{Y}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{X}_{i,:}\mathbf{W} - \mathbf{Y}_{i,:}\|_2^2 \qquad \triangleright \text{Frobenius norm}$$

where the Frobenius norm is defined as $\|\mathbf{A}\|_F^2 = \sum_{ij} A_{ij}^2$. The resulting solution is

$$\mathbf{W}_{\text{MLE}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y}.$$

We leave this derivation as an exercise, using partial derivatives or matrix derivative rules.

Exercise 14: Derive this linear regression solution for multiple outputs. □

Looking at this solution, we can see that it is actually equivalent to computing a separate linear regression solution for each output separately. Namely, for $\mathbf{y}_k \in \mathbb{R}^n$ the k-th target for each sample, with $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_m] \in \mathbb{R}^{n \times m}$,

$$\begin{aligned}\mathbf{W}_{\text{MLE}} &= (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_m] = [(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}_1, (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}_2, \ldots, (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}_m] \\ &= [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m]\end{aligned}$$

where $\mathbf{w}_k = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}_k$ the linear regression solution for scalar target y_k in the target vector $\mathbf{y} = [y_1, y_2, \ldots, y_m]$.

This result is almost disappointing: shouldn't learning all of the targets at once be more useful than simply learning than separately? The reason is that we do not constrain the models to consider relationships between the targets. In the absence of such constraints, the best way to minimize the squared error is to get the best linear fit for each scalar target. We can impose constraints on $\mathbf{W}$ so that some of the weights must be shared between targets. For linear regression, one way to do this is called *reduced rank regression*. When we move to learning features in Chapter 9, we will see another way to encourage solutions to jointly consider the targets.

3.2 Stability and the Bias-Variance of the OLS Solution

The OLS solution can be unstable. In this section, we show why this is the case, and discuss how regularization can be used to mitigate this problem. We will then revisit the bias-variance trade-off, and discuss the bias and variance of the OLS solution.

3.2.1 Sensitivity of the OLS solution

The OLS solution is unstable if $\mathbf{X}^\top \mathbf{X}$ is not invertible. This can occur for two main reasons: linearly dependent features and small datasets. Data sets often include large numbers of features, which are sometimes linearly dependent or highly correlated (linear dependence except for noise). If the dataset is small, it is feasible that some features are the same across samples, again resulting in low-rank $\mathbf{X}$. When $\mathbf{X}^\top \mathbf{X}$ is not invertible—or ill-conditioned—the OLS solution is highly sensitive to small perturbations in $\mathbf{y}$ and $\mathbf{X}$.

To see why, we will look at the singular value decomposition (SVD) of $\mathbf{X}$. As with the previous linear algebra constructs, it allows us to easily examine properties of $\mathbf{X}$. We overviewed the SVD in Section 1.5.2. Let's consider the common case, where $n > d$: the number of samples is greater than the input dimension. The singular value decomposition of $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ for orthonormal matrices $\mathbf{U} \in \mathbb{R}^{n \times n}, \mathbf{V} \in \mathbb{R}^{d \times d}$ and non-negative (rectangular) diagonal matrix $\mathbf{\Sigma} \in \mathbb{R}^{n \times d}$. The diagonal entries in $\mathbf{\Sigma}$ are the singular values, which we

typically order in descending order $\sigma_1, \sigma_2, \ldots, \sigma_d$, giving

$$\boldsymbol{\Sigma} \stackrel{\text{def}}{=} \begin{bmatrix} \sigma_1 & 0 & 0 & \ldots & 0 \\ 0 & \sigma_2 & 0 & \ldots & 0 \\ & & \vdots & & \\ 0 & 0 & \ldots & 0 & \sigma_d \\ 0 & 0 & \ldots & 0 & 0 \\ & & \vdots & (n-d) \text{ rows} & \text{of zeros} \\ 0 & 0 & \ldots & 0 & 0 \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Sigma}_d \\ \mathbf{0} \end{bmatrix} \qquad \text{where } \boldsymbol{\Sigma}_d \stackrel{\text{def}}{=} \begin{bmatrix} \sigma_1 & 0 & \ldots & 0 \\ 0 & \sigma_2 & \ldots & 0 \\ & & \vdots & \\ 0 & 0 & \ldots & \sigma_d \end{bmatrix}.$$

The matrix $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_n] \in \mathbb{R}^{n \times n}$ is the orthonormal matrix composed of the left singular vectors, and $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_d] \in \mathbb{R}^{d \times d}$ is the orthonormal matrix composed of the right singular vectors. Recall that we can equivalently write this with the thin SVD, assuming $n > d$, using $\mathbf{U}_d = [\mathbf{u}_1, \ldots, \mathbf{u}_d] \in \mathbb{R}^{n \times d}$ the first d left singular vectors: $\mathbf{X} = \mathbf{U}_d \boldsymbol{\Sigma}_d \mathbf{V}^\top$. This is because the zeros in $\boldsymbol{\Sigma}$ multiply the columns in $\mathbf{U}$ at positions $d+1, \ldots, n$. Nonetheless, we still write the full SVD because it will be easier to deal with orthonormal $\mathbf{U}$ than the non-square $\mathbf{U}_d$, at least to start.

Now we can discuss the least-squares solution, in terms of the singular value decomposition of $\mathbf{X}$. Notice that

$$\mathbf{X}^\top \mathbf{X} = \mathbf{V} \boldsymbol{\Sigma}^\top \mathbf{U}^\top \mathbf{U} \boldsymbol{\Sigma} \mathbf{V}^\top = \mathbf{V} \boldsymbol{\Sigma}_d^2 \mathbf{V}^\top$$

because $\mathbf{U}$ is orthonormal and so $\mathbf{U}^\top \mathbf{U} = \mathbf{I}$ the identity matrix ($\mathbf{I}$ is a diagonal matrix with ones on the diagonal) and because

$$\boldsymbol{\Sigma}^\top \boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_d & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Sigma}_d \\ \mathbf{0} \end{bmatrix} = \boldsymbol{\Sigma}_d \boldsymbol{\Sigma}_d + \mathbf{0}\mathbf{0} = \boldsymbol{\Sigma}_d^2.$$

The inverse of $\mathbf{X}^\top \mathbf{X}$ exists if $\mathbf{X}$ is full rank, i.e., $\boldsymbol{\Sigma}_d$ has no zeros on the diagonal, because $(\mathbf{X}^\top \mathbf{X})^{-1} = \mathbf{V} \boldsymbol{\Sigma}_d^{-2} \mathbf{V}^\top$. The resulting solution for $\mathbf{w}$ looks like[1]

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} = \mathbf{V} \boldsymbol{\Sigma}_d^{-1} \mathbf{U}_d^\top \mathbf{y} = \sum_{j=1}^{d} \frac{\mathbf{u}_j^\top \mathbf{y}}{\sigma_j} \mathbf{v}_j \tag{3.2}$$

The solution in Equation (3.3) makes it clear why the linear regression solution can be sensitive to perturbations. For small singular values, σ_j^{-1} is large and amplifies any changes in $\mathbf{y}$. For example, for slightly different noise component ϵ_i for the ith sample, the solution vector $\mathbf{w}$ could be very different. A common strategy to deal with this instability is to drop or truncate small singular values. This is a form of regularization.

Remark: In the general case, where $\mathbf{X}$ is not full rank, we can still obtain a least-squares solution to $\mathbf{X}^\top \mathbf{X} \mathbf{w} = \mathbf{X}^\top \mathbf{y}$. Now, there are potentially infinitely many solutions. The common choice is to select the minimum variance solution, which corresponds to dropping the components (singular vectors) for the zero singular values:

$$\mathbf{w} = \sum_{j=1}^{\text{rank of } \mathbf{X}} \frac{\mathbf{u}_j^\top \mathbf{y}}{\sigma_j} \mathbf{v}_j. \tag{3.3}$$

[1] The last step in the below equation, writing the matrix product as a sum, is not immediately obvious. But it is not hard to find by simply multiplying out each matrix-vector product: first $\mathbf{a} = \mathbf{U}_d^\top \mathbf{y}$, then $\mathbf{b} = \boldsymbol{\Sigma}_d^{-1} \mathbf{a}$ and finally $\mathbf{V}\mathbf{b}$. As an exercise, see if you can derive this last equality.

Example 4: [Dependent or correlated features] Let's look at a simple example of why $\mathbf{X} \in \mathbb{R}^{n \times d}$ might have small singular values. First, assume $d = 2$ and $x_2 = x_1$, i.e., that the second features is a copy of the first and simply redundant. Then $\mathbf{X} = \mathbf{U}_2\mathbf{\Sigma}_2\mathbf{V}^\top$ is the thin SVD of $\mathbf{X}$, where $\mathbf{U}_2$ only has the first two columns of the full SVD. We can write this thin SVD because $\mathbf{X} = \mathbf{U}_2\mathbf{\Sigma}_2\mathbf{V}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where the zero singular values zero out the remaining columns of $\mathbf{U}$.

The SVD of just the first column $\mathbf{a}_1 \in \mathbb{R}^{n \times 1}$ is straightforward: $\mathbf{a}_1 = \mathbf{u}_1\sigma_1 v_1$, where $\mathbf{u}_1 = \mathbf{a}_1/\|\mathbf{a}_1\|$, $\sigma_1 = \|\mathbf{a}_1\|$ and $v_1 = 1$. The SVD of $\mathbf{X} = [\mathbf{a}_1 \quad \mathbf{a}_2]$ is therefore, for any n-dimensional unit vector $\mathbf{u}_2$ that is orthogonal to $\mathbf{u}_1$, and right singular vectors $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^2$,

$$\mathbf{X} = [\mathbf{u}_1 \quad \mathbf{u}_2]\mathbf{\Sigma}[\mathbf{v}_1 \quad \mathbf{v}_2]^\top = [\mathbf{u}_1 \quad \mathbf{u}_2] \begin{bmatrix} 1.414\sigma_1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \sqrt{0.5} & \sqrt{0.5} \\ -\sqrt{0.5} & \sqrt{0.5} \end{bmatrix} = \mathbf{u}_1\sigma_1[1.0 \quad 1.0]$$

where we extended v_1 to two-dimensions (since $d = 2$), and defined $\mathbf{v}_2$ to be orthogonal to that vector, and had to rescale σ_1 by $1/\sqrt{0.5} \approx 1.414$ to maintain unit singular vectors. So because $\mathbf{a}_2$ is dependent on $\mathbf{a}_1$, the rank does not increase when we add it as a column and the singular value $\sigma_2 = 0$.

If instead $\mathbf{a}_2 = \mathbf{a}_1 + \epsilon$ for a small noise vector $\epsilon \in \mathbb{R}^n$ (linearly correlated), then instead we would find that σ_2 would no longer be zero, but would be very close to zero, because $\mathbf{u}_1$ and the first singular value σ_1 would largely be able to recreate $\mathbf{a}_2$. □

This example highlights that the rank of $\mathbf{X}$ might be lower if there is redundancy in the features. Similarly, we can have an $\mathbf{X}$ with nearly zero singular values if there is redundancy in the training samples. Even with $n > d$, for n close to d, two input feature vectors could accidentally be similar; they are randomly sampled, after all. This is why in linear regression overfitting can manifest in large weights. It is actually fitting to the noise, such as the ϵ in the above example, instead of to actual patterns. This noise is recognizable by the fact that the singular values are very small, resulting in large weights.

3.2.2 Improving Stability with ℓ_2 Regularization

The OLS solution is the maximum likelihood solution. But, we can instead use a MAP objective. Here we discuss ℓ_2 regularization—which corresponds to a Gaussian prior—which helps improve the stability of the solution.

Let's use the zero-mean Gaussian prior, $\mathcal{N}(\mathbf{0}, \sigma^2\lambda^{-1}\mathbf{I})$, where we pick regularization parameter $\lambda > 0$. The choice of variance $\sigma^2\lambda^{-1}\mathbf{I}$ will be made clear below, but intuitively we scale the regularization to be higher if the variance in targets is higher. To write down the log of the posterior, we need the log of the likelihood and the log of the prior. We have already taken the log for the MLE solution, so lets focus on the log of the prior. Then

$$-\ln p(\mathbf{w}) = \tfrac{1}{2}\ln(2\pi|\sigma^2\lambda^{-1}\mathbf{I}|) + \frac{\mathbf{w}^\top\mathbf{w}}{2\sigma^2\lambda^{-1}} = \tfrac{1}{2}\ln(2\pi) - d\ln(\lambda/\sigma^2) + \frac{\lambda}{2\sigma^2}\mathbf{w}^\top\mathbf{w}$$

because $|\sigma^2\lambda^{-1}\mathbf{I}| = (\sigma^2/\lambda)^d$, where $|\mathbf{A}|$ is the determinant of the matrix $\mathbf{A}$. We can drop the first constant which does not affect the selection of $\mathbf{w}$.

Now we can combine the negative log-likelihood and the negative log prior. Then ignor-

ing constants, we can add up the negative log-likelihood and negative log prior to get

$$\begin{aligned}\operatorname*{argmin}_{\mathbf{w}} -\ln(p(\mathbf{y}|\mathbf{X},\mathbf{w})) - \ln p(\mathbf{w}) &= \operatorname*{argmin}_{\mathbf{w}} \frac{1}{2\sigma^2}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 + \frac{\lambda}{2\sigma^2}\mathbf{w}^\top\mathbf{w} \\ &= \operatorname*{argmin}_{\mathbf{w}} \frac{1}{2}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 + \sigma^2\frac{\lambda}{2\sigma^2}\mathbf{w}^\top\mathbf{w}\end{aligned}$$

where the second line follows from multiplying both the first and second term by σ^2. Therefore if we assume that the weights have a zero-mean Gaussian prior $\mathcal{N}(\mathbf{0}, \lambda^{-1}\sigma^2\mathbf{I})$, then we get the following ridge regression problem:

$$c(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2.$$

The idea is to penalize weight coefficients that are too large. The larger the λ, the more large weights are penalized. Correspondingly, larger λ corresponds to a smaller covariance in the prior, pushing the weights to stay near zero. The MAP estimate, therefore, has to balance between this prior on the weights, and fitting the observed data.

If we solve this equation in a similar manner as before, we obtain

$$\mathbf{w}_{\text{MAP}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}.$$

This has the nice effect of shifting the squared singular values in Σ_d^2 by λ, removing stability issues with dividing by small singular values, as long as λ is itself large enough. In particular,

$$\mathbf{w}_{\text{MAP}} = \sum_{j=1}^{d} \frac{\sigma_j}{\sigma_j^2+\lambda}\mathbf{u}_j^\top\mathbf{y}\mathbf{v}_j. \tag{3.4}$$

Notice that when $\lambda = 0$, then we have $\sigma_j/\sigma_j^2 = \sigma_j^{-1}$, which is the solution we found for MLE. With $\lambda > 0$, we ensure that we do not divide by very small singular values σ_j^2, and so improve stability. In the next section, we discuss how this can reduce the variance of the solution significantly, albeit with some introduction of bias.

3.2.3 The Bias-Variance Trade-off

A natural question to ask is how this regularization parameter can be selected, and the impact on the final solution vector. The selection of this regularization parameter leads to a bias-variance trade-off. To understand this trade-off, we need to understand what it means for the solution to be biased, and how to characterize the variance of the solution, across possible datasets.

Let us begin by presuming that the distributional assumptions behind linear regression are true. This means that there exists a true parameter $\boldsymbol{\omega}$ such that for each of the data points $Y_i = \sum_{j=1}^{d}\omega_j X_{ij} + \varepsilon_i$, where the ε_j are i.i.d. random variables drawn according to $\mathcal{N}(0,\sigma^2)$. We can characterize the solution vector (estimator) $\mathbf{w}_{\text{MLE}}$ as a random variable, where the randomness is across possible datasets that could have been observed. In this sense, we are considering the dataset $\mathcal{D}$ to be a random variable, and the solution $w_{\text{MLE}}(\mathcal{D})$ from that dataset as a function of this random variable.

The reason we care about the bias and variance of $\mathbf{w}_{\text{MLE}}$ is because the expected mean-squared error to the true weights can be decomposed into the bias and variance.

$$\mathbb{E}\left[\|\mathbf{w}(\mathcal{D}) - \boldsymbol{\omega}\|_2^2\right] = \mathbb{E}\left[\sum_{j=1}^{d}(w_j(\mathcal{D}) - \omega_j)^2\right] = \sum_{j=1}^{d}\mathbb{E}\left[(w_j(\mathcal{D}) - \omega_j)^2\right]$$

where we can then further simplify

$$\mathbb{E}\left[(w_j(\mathcal{D}) - \omega_j)^2\right] = \underbrace{(\mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j)}_{\text{Bias}}{}^2 + \text{Var}\left[w_j(\mathcal{D})\right]$$

(For all these steps, see Appendix A.1.1).

The bias-variance trade-off reflects the fact that we could potentially reduce the mean-squared error by incurring some bias, as long as the variance is decreased more than the squared bias. Note that we do not directly optimize the bias-variance trade-off. We cannot actually measure the bias, so we do not directly minimize these terms. Rather, this decomposition guides how we select model classes.

We can use this formula to contrast the bias and variance for the OLS and ℓ_2-regularized solutions. What we find (again, see Appendix A.1.1 for the derivations) is that

$$\mathbb{E}[w_{\text{MLE}}(\mathcal{D})] = \boldsymbol{\omega}$$
$$\sum_{j=1}^{d}\text{Var}\left[w_{\text{MLE},j}(\mathcal{D})\right] = \sigma^2\mathbb{E}\left[\sum_{j=1}^{d}\sigma_j^{-2}\right]$$

An estimator whose expected value is the true value of the parameter is called an *unbiased estimator*. So $w_{\text{MLE}}(\mathcal{D})$ is an unbiased estimator, but unfortunately it can have high variance. This formula makes it clear that the variance of the weights is tied to the magnitudes of the inverse of the singular values. If we have very small singular values, then this sum is much larger. The singular values will not be small for all datasets that could have been observed, but in cases where overfitting is possible (small n), we expect it to happen for a large proportion of datasets.

The regularized solution, on the other hand, is much less likely to have high covariance, but will no longer be unbiased. Let $w_{\text{MAP}}(\mathcal{D})$ be the MAP estimate for the ℓ_2 regularized problem with $\lambda > 0$. The expected value of $w_{\text{MAP}}(\mathcal{D})$ is

$$\mathbb{E}[w_{\text{MAP}}(\mathcal{D})] = \mathbb{E}\left[\left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)^{-1}(\mathbf{X}^\top\mathbf{X})\boldsymbol{\omega}\right] \neq \boldsymbol{\omega}.$$

As $\lambda \to 0$, the MAP solution gets closer and closer to being unbiased. But, the variance also decreases with larger λ. Specifically, we have

$$\sum_{j=1}^{d}\text{Var}\left[w_{\text{MAP},j}(\mathcal{D})\right] = \sigma^2\mathbb{E}\left[\sum_{j=1}^{d}\frac{\sigma_j^2}{(\sigma_j + \lambda)^2}\right]$$

We now have $\sigma_d^2 + \lambda$ in the denominator, which is not that small if λ is not that small. Consequently, we expect w_{MAP} to have lower variance across different datasets. This correspondingly implies that we are less likely to overfit to any one dataset. Notice that as

$\lambda \to \infty$, the variance decreases to zero, but the bias increases to its maximal value (i.e., the norm of the true weights). There is an optimal choice of λ—not too big and not too small—that minimizes this bias-variance trade-off—if we could find it.

Exercise 15: Derive the covariance formula for $w_{\text{MAP}}(\mathcal{D})$. □

Exercise 16: Recall that for polynomial regression we first transformed the inputs into new polynomial features. Then, we simply treated this new transformed dataset as a linear regression problem, though we know now that we are learning a nonlinear predictor in the original space. Let us think of $\mathbf{\Phi}$ as the transformed space, namely consisting of the polynomial features. This new matrix has a much larger d—many more columns—since we expanded the number of features. Do you think this $\mathbf{\Phi}$ is more or less likely to suffer from having small singular values, than the original one $\mathbf{X}$ before the transformation? □

The above discussion assumes *realizability*, namely that the true model is linear. In practice, we not only have bias from ℓ_2 regularization but also due to the fact that we likely do not have the true model in our model class. For example, we might be using linear functions, when the true model is from a 9th order polynomial or a neural network or even some function it is hard for us to represent. We will revisit this non-realizable case, and generalization error, when we move to more complex models that learn data representations.

Chapter 4

Multivariate Optimization Principles

You have learned the basics of gradient descent and stochastic gradient descent. Reread the short Chapter 6 in those previous notes as a quick refresher. In this chapter, we will discuss the second-order gradient descent update for the multivariate case. We start by re-deriving the second-order gradient descent update rule, now for the multivariate setting, and introduce the Hessian matrix. We will provide new stepsize selection algorithms, expanding on the basic heuristics you have already seen.

4.1 Second-order Multivariate Gradient Descent

We can generalize the discussion on obtaining the gradient descent update from the univariate case to the multivariate case using the multivariate Taylor series approximation. The second-order Taylor approximation for a real-valued function of multiple variables can be written as

$$c(\mathbf{w}) \approx \hat{c}(\mathbf{w}) = c(\mathbf{w}_0) + \nabla c(\mathbf{w}_0)^\top (\mathbf{w} - \mathbf{w}_0) + \frac{1}{2} (\mathbf{w} - \mathbf{w}_0)^\top \mathbf{H}_{c(\mathbf{w}_0)} (\mathbf{w} - \mathbf{w}_0),$$

where

$$\nabla c(\mathbf{w}_0) = \left(\frac{\partial c}{\partial w_1}(\mathbf{w}_0), \frac{\partial c}{\partial w_2}(\mathbf{w}_0), ..., \frac{\partial c}{\partial w_d}(\mathbf{w}_0) \right)^\top \quad \in \mathbb{R}^d$$

is the gradient of function c evaluated at $\mathbf{w}_0$ and

$$\mathbf{H}_{c(\mathbf{w}_0)} = \begin{bmatrix} \frac{\partial^2 c}{\partial w_1^2}(\mathbf{w}_0) & \frac{\partial^2 c}{\partial w_1 \partial w_2}(\mathbf{w}_0) & \cdots & \frac{\partial^2 c}{\partial w_1 \partial w_d}(\mathbf{w}_0) \\ \frac{\partial^2 c}{\partial w_2 \partial w_1}(\mathbf{w}_0) & \frac{\partial^2 c}{\partial w_2^2}(\mathbf{w}_0) & & \\ \vdots & \vdots & \ddots & \\ \frac{\partial^2 c}{\partial w_d \partial w_1}(\mathbf{w}_0) & \cdots & & \frac{\partial^2 c}{\partial w_d^2}(\mathbf{w}_0) \end{bmatrix} \quad \in \mathbb{R}^{d \times d}$$

is the Hessian matrix of function c evaluated at $\mathbf{w}_0$.

Example 5:

Let us consider a two-dimensional example, for the surface depicted in Figure 4.2. This example corresponds to a squared linear regression objective with one sample (x, y), $c(\mathbf{w}) = \frac{1}{2}(w_1 + xw_2 - y)^2$ for $x = 0.1$ and $y = 2.0$. The gradient at some point $\mathbf{w}$ is

$$\nabla c(\mathbf{w}) = \begin{bmatrix} \frac{\partial c}{\partial w_1}(\mathbf{w}) \\ \frac{\partial c}{\partial w_2}(\mathbf{w}) \end{bmatrix} = \begin{bmatrix} (w_1 + xw_2 - y) \\ (w_1 + xw_2 - y)x \end{bmatrix}$$

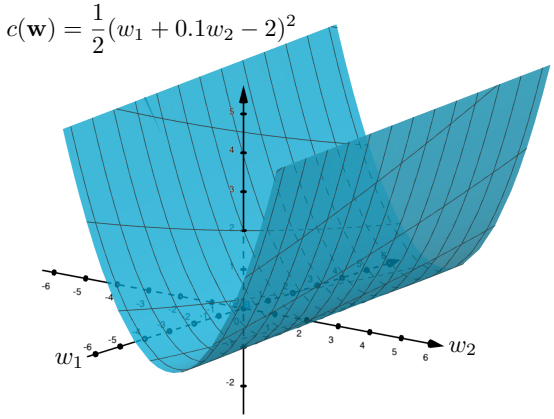


Figure 4.1: The loss landscape for $\mathbf{w} \in \mathbb{R}^2$, with one dimension being very flat and the other very curved. This example corresponds to a squared linear regression objective with one sample (x, y), $c(\mathbf{w}) = \frac{1}{2}(w_1 + xw_2 - y)^2$ for $x = 0.1$ and $y = 2.0$.

and the Hessian is

$$\mathbf{H}_{c(\mathbf{w})} = \begin{bmatrix} \frac{\partial^2 c}{\partial w_1^2}(\mathbf{w}) & \frac{\partial^2 c}{\partial w_1 \partial w_2}(\mathbf{w}) \\ \frac{\partial^2 c}{\partial w_2 \partial w_1}(\mathbf{w}) & \frac{\partial^2 c}{\partial w_2^2}(\mathbf{w}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial w_1}(w_1 + xw_2 - y) & \frac{\partial}{\partial w_1}(w_1 + xw_2 - y)x \\ \frac{\partial}{\partial w_2}(w_1 + xw_2 - y) & \frac{\partial}{\partial w_2}(w_1 + xw_2 - y)x \end{bmatrix}$$
$$= \begin{bmatrix} 1 & x \\ x & x^2 \end{bmatrix}$$

For the given sample $x = 0.1$ and $y = 2.0$, we have that

$$\nabla c(\mathbf{w}) = \begin{bmatrix} (w_1 + 0.1w_2 - 2.0) \\ 0.1(w_1 + 0.1w_2 - 2.0) \end{bmatrix} \quad \mathbf{H}_{c(\mathbf{w})} = \begin{bmatrix} 1 & 0.1 \\ 0.1 & 0.01 \end{bmatrix}$$

The gradient is local, around a specific point $\mathbf{w}_0$. If we have $\mathbf{w}_0 = (0, 0)$, then

$$\nabla c(\mathbf{w}_0) = \begin{bmatrix} (0 + 0.1 \times 0 - 2.0) \\ 0.1(0 + 0.1 \times 0 - 2.0) \end{bmatrix} = \begin{bmatrix} -2 \\ -0.2 \end{bmatrix}$$

This gradient points in an ascent direction. For this example, the Hessian is the same for every $\mathbf{w}$, but this is not always the case. Usually, it changes depending on where we are on the loss landscape. □

We provide some intuition for the Hessian in the next section, but here it can be intuitively considered analogous to the second derivative. Like the second derivative, it provides information about the curvature of the function, and so provides useful information about how much to step in the direction of the gradient for each w_i.

As a reminder about matrix-vector multiplication, the product of a $d \times d$ matrix $\mathbf{H}$ and $d \times 1$ vector $\mathbf{w}$ is a $d \times 1$ vector $\mathbf{Hw}$. Then, taking $\mathbf{w}^\top \mathbf{Hw}$ is the dot product between a $1 \times d$

vector $\mathbf{w}^\top$ and $d \times 1$ vector $\mathbf{Hw}$, resulting in a scalar. For matrix-vector multiplication,

$$\mathbf{Hw} = \begin{bmatrix} \mathbf{H}_{1:} \\ \mathbf{H}_{2:} \\ \vdots \\ \mathbf{H}_{d:} \end{bmatrix} \mathbf{w} = \begin{bmatrix} \mathbf{H}_{1:}\mathbf{w} \\ \mathbf{H}_{2:}\mathbf{w} \\ \vdots \\ \mathbf{H}_{d:}\mathbf{w} \end{bmatrix} = \begin{bmatrix} \langle \mathbf{H}_{1:}, \mathbf{w} \rangle \\ \langle \mathbf{H}_{2:}, \mathbf{w} \rangle \\ \vdots \\ \langle \mathbf{H}_{d:}, \mathbf{w} \rangle \end{bmatrix}$$

When performing matrix-vector multiplication, you can just imagine the vector $\mathbf{w}$ turning sideways and multiplying each row of $\mathbf{H}$. For matrix-matrix multiplication, $\mathbf{AB}$, you have to ensure that the second dimension of $\mathbf{A}$ equals the first dimension of $\mathbf{B}$. The matrix-matrix multiplication decomposes into matrix-vector multiplication, for each column of $\mathbf{B}$.

As before, to get the incremental update, we can take the gradient of this approximation and obtain the (local) stationary point. Using the basic rules summarized in Section 1.5.3, the gradient of $\hat{c}(\mathbf{w})$ is

$$\nabla \hat{c}(\mathbf{w}) = \nabla c(\mathbf{w}_0) + \mathbf{H}_{c(\mathbf{w}_0)} (\mathbf{w} - \mathbf{w}_0) .$$

Again, we want to find $\mathbf{w}_1$ such that this gradient is zero. To solve for $\mathbf{H}_{c(\mathbf{w}_0)} (\mathbf{w} - \mathbf{w}_0) = -\nabla c(\mathbf{w}_0)$, one can compute the inverse $\mathbf{H}^{-1}_{c(\mathbf{w}_0)}$ and multiply both sides of the equation by this inverse. This is again analogous to the inverse of a scalar: $h^{-1}h = 1$. The corresponding multivariate update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \left(\mathbf{H}_{c(\mathbf{w}_t)}\right)^{-1} \nabla c(\mathbf{w}_t). \tag{4.1}$$

In Equation 4.1, both gradient and Hessian are evaluated at point $\mathbf{w}_t$.

4.2 Visualizing the Hessian

Like the second-derivative, the Hessian reflects the curvature of the function at the point $\mathbf{w}_0$. Each entry reflects how the partial derivative for w_j changes when w_i is changed. For additional intuition, consider the directional derivative. The directional derivative reflects how a (multivariate) function changes when stepping a small amount τ in some fixed direction $\mathbf{u}$

$$\lim_{\tau \to 0} \frac{c(\mathbf{w} + \tau \mathbf{u}) - c(\mathbf{w})}{\tau}.$$

Once we restrict ourselves to how the function changes in this one direction, it is easier to imagine and it allows us to use the familiar second derivative test for the univariate setting.

Assume that $\mathbf{w}$ is a stationary point, namely that $\nabla c(\mathbf{w}) = \mathbf{0}$. We would like to understand if we have a local minima, local maxima or saddlepoint. Let

$$\begin{aligned} \mathbf{w}(\tau) &= \mathbf{w} + \tau \mathbf{u} \\ g(\tau) &= c(\mathbf{w}(\tau)). \end{aligned}$$

We can use the chain rule on $g(\tau)$ to compute the derivative w.r.t. τ.

$$g'(\tau) = \nabla c(\mathbf{w}(\tau))^\top \frac{\partial (\mathbf{w}(\tau))}{\partial \tau} = \nabla c(\mathbf{w}(\tau))^\top \mathbf{u}$$

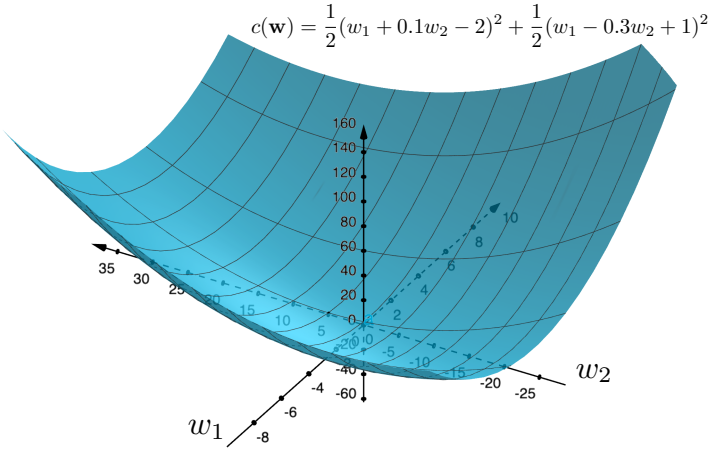


Figure 4.2: The loss landscape for $\mathbf{w} \in \mathbb{R}^2$, with one dimension being quite flat and the other quite curved. This example corresponds to a squared linear regression objective with two samples $(x_1, y_1) = (0.1, 2)$ and $(x_2, y_2) = (-0.3, -1)$, giving $c(\mathbf{w}) = \frac{1}{2}(w_1 + 0.1w_2 - 2)^2 + \frac{1}{2}(w_1 - 0.3w_2 + 1)^2$.

Therefore, we can use this generic gradient, and evaluate at $\tau = 0$

$$g'(0) = \nabla c(\mathbf{w}(0))^\top \mathbf{u} = \nabla c(\mathbf{w})^\top \mathbf{u} = 0$$

where the last equality occurs because $\mathbf{w}$ is a stationary point and so $\nabla c(\mathbf{w}) = \mathbf{0}$. The second derivative is

$$g''(\tau) = \frac{\partial(\mathbf{w}(\tau))}{\partial \tau}^\top \mathbf{H}_{c(\mathbf{w}(\tau))} \frac{\partial(\mathbf{w}(\tau))}{\partial \tau} = \mathbf{u}^\top \mathbf{H}_{c(\mathbf{w}(\tau))} \mathbf{u}$$
$$g''(0) = \mathbf{u}^\top \mathbf{H}_{c(\mathbf{w})} \mathbf{u}$$

For this stationary point $\mathbf{w}$ (corresponding to $\tau = 0$) to be a local minimum, $g''(0)$ has to satisfy the second derivative test: $g''(0) > 0$. This test is only satisfied if $\mathbf{H}_{c(\mathbf{w})}$ is positive definite, by definition of a positive definite matrix. Recall that a positive-definite matrix $\mathbf{H}$ is one for which, given any $\mathbf{u} \neq \mathbf{0}$, $\mathbf{u}^\top \mathbf{H} \mathbf{u} > 0$, or equivalently, has all eigenvalues greater than zero. Since $\mathbf{u}$ was an arbitrary direction away from $\mathbf{w}$, the Hessian must be positive-definite to ensure that $g''(0) > 0$ for all $\mathbf{u} \neq \mathbf{0}$.

The eigenvalues of the Hessian, therefore, reflect the curvature of the function locally. If $\mathbf{H}_{c(\mathbf{w})}$ has a very small eigenvalue λ_j, then the corresponding eigenvector $\mathbf{u}_j$—satisfying $\mathbf{H}_{c(\mathbf{w})} \mathbf{u}_j = \lambda_j \mathbf{u}_j$—is a direction away from $\mathbf{w}$ where the function is almost flat. This is because $g''(0) = \mathbf{u}_j^\top \mathbf{H}_{c(\mathbf{w})} \mathbf{u}_j = \lambda_j \|\mathbf{u}_j\|_2^2 = \lambda_j$ is very small.

Example 6: Let us consider a similar example to the one above, but now with two samples. In addition to $(x_1, y_1) = (0.1, 2)$, we also see $(x_2, y_2) = (-0.3, -1)$. The objective is

$$c(\mathbf{w}) = \frac{1}{2}(\underbrace{w_1 + x_1 w_2 - y_1}_{\delta_1})^2 + \frac{1}{2}(\underbrace{w_1 + x_2 w_2 - y_2}_{\delta_2})^2$$

The gradient is

$$\nabla c(\mathbf{w}) = \delta_1 \begin{bmatrix} 1 \\ x_1 \end{bmatrix} + \delta_2 \begin{bmatrix} 1 \\ x_2 \end{bmatrix}$$

and the Hessian is

$$\mathbf{H}_{c(\mathbf{w})} = \begin{bmatrix} 1 & x_1 \\ x_1 & x_1^2 \end{bmatrix} + \begin{bmatrix} 1 & x_2 \\ x_2 & x_2^2 \end{bmatrix} = \begin{bmatrix} 2 & -0.2 \\ -0.2 & 0.1 \end{bmatrix}$$

The eigenvalue decomposition of the Hessian will let us see how it impacts the update. Using a numerical library, we can find that $\mathbf{H}_{c(\mathbf{w})} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$ for $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2]$ with eigenvectors $\mathbf{u}_1, \mathbf{u}_2$ as the columns

$$\mathbf{u}_1 \approx \begin{bmatrix} -1 \\ 0.1 \end{bmatrix} \qquad \mathbf{u}_2 \approx \begin{bmatrix} 0.1 \\ 0.95 \end{bmatrix}$$

and eigenvalues $\lambda_1 \approx 6.5$ and $\lambda_2 \approx 0.08$. The inverse of the Hessian is

$$\mathbf{H}^{-1}_{c(\mathbf{w})} = \mathbf{U} \begin{bmatrix} 1/\lambda_1 & 0 \\ 0 & 1/\lambda_2 \end{bmatrix} \mathbf{U}^\top = \begin{bmatrix} 0.625 & 1.25 \\ 1.25 & 12.5 \end{bmatrix}$$

The Hessian shows that we have one direction with steep curvature ($\lambda_1 \approx 6.5$) and the other being very flat ($\lambda_2 \approx 0.08$). We can see this visually, but in higher dimensions the easier way to see it is through these eigenvalues. □

Exercise 17: If we had taken the eigenvalue decomposition from Example 5, then we would have found that the eigenvalues are $\lambda_1 = 101/100$ and $\lambda_2 = 0$. An eigenvalue that is zero indicates a perfectly flat function in one direction. Why did this happen? Also notice that in this situation, we cannot take the inverse of the Hessian, since it is not invertible! □

Example 7: The Hessian for linear regression is

$$H_{c(\mathbf{w})} = 2\mathbf{X}^\top\mathbf{X}.$$

This matrix is the same regardless of which point $\mathbf{w}$ we query the Hessian for. Recall that the matrix $\mathbf{X}^\top\mathbf{X} \in \mathbb{R}^{d\times d}$ reflects the (sample) covariance of inputs, because $\mathbf{X}^\top\mathbf{X} = \sum_{i=1}^n \mathbf{x}_i\mathbf{x}_i^\top$. To determine if the stationary point for linear regression is a (local) minimum, we have the check if the Hessian is positive definite. Consider that for any vector $\mathbf{w} \neq \mathbf{0}$,

$$\mathbf{w}^\top\mathbf{X}^\top\mathbf{X}\mathbf{w} = (\mathbf{X}\mathbf{w})^\top\mathbf{X}\mathbf{w} = \|\mathbf{X}\mathbf{w}\|_2^2 \geq 0$$

where equality with zero can only happen—for some $\mathbf{w}$—if the columns of $\mathbf{X}$ are linearly dependent. Since the Hessian is positive semi-definite for every $\mathbf{w}$, this verifies the convexity of $c(\mathbf{w})$. Furthermore, if the columns of $\mathbf{x}$ are linearly independent, the Hessian is positive definite, which implies that the global minimum is unique. □

Exercise 18: What is the computational complexity of computing $\mathbf{X}^\top\mathbf{X}$? □

Exercise 19: In Section 2.4, we found the MLE solution $\boldsymbol{\mu}^*$ for a multivariate Gaussian. We claimed the objective was convex, so we knew that the solution was a global minimum. Alternatively, we could haved check that it is a local minimum using the second derivative test, by checking if the Hessian is positive definite. If it is, then because we have a single stationary point, we know that we have a global minimum.

Show that the Hessian for $\boldsymbol{\mu}^*$ is positive definite. Let $\tilde{c}(\boldsymbol{\mu}) = c(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ for any positive definite $\boldsymbol{\Sigma}$ and note that the Hessian is $\mathbf{H}_{\tilde{c}(\boldsymbol{\mu}^*)} = \boldsymbol{\Sigma}^{-1}$. Hint: recall that we can write $\boldsymbol{\Sigma} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top$ using it's eigenvalue decomposition and $\boldsymbol{\Sigma}^{-1} = \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{U}^\top$. Since $\boldsymbol{\Sigma}$ is positive definite, we know that the eigenvalues on the diagonal in $\boldsymbol{\Lambda}$ are positive. □

4.3 Contrasting Convergence Rates

The size of the Hessian makes the choice between first-order and second-order gradient descent less obvious in the multivariate case than in the univariate case. In the univariate (scalar) setting, as long as we have a formula for the second derivative, then it is likely efficient to compute and provides a good choice for the stepsize. In the multivariate setting, computing the Hessian itself is expensive (quadratic in the size of $\mathbf{w}$) and it is further even more expensive to compute the inverse of the Hessian. For example, if computing the Hessian costs $O(d^2n)$ as it does for the linear regression objective, then the computational complexity of the second-order gradient descent is $O(d^3 + d^2n)$ in each iteration, assuming $O(d^3)$ time for finding matrix inverses. On the other hand, again for linear regression, the computational complexity for first-order gradient descent is only $O(dn)$ per iteration.

The first order update for the multivariate case is an even greater approximation, because the whole Hessian is approximated with a scalar $\frac{1}{\eta_t}$ (making the Hessian approximation a diagonal matrix with $\frac{1}{\eta_t}$ on the diagonal). The gradient of the first-order approximation is

$$\nabla \hat{c}(\mathbf{w}) = \nabla c(\mathbf{w}_0) + \frac{1}{\eta_t}(\mathbf{w} - \mathbf{w}_0)$$

and the resulting first-order update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla c(\mathbf{w}_t). \tag{4.2}$$

We can do a little better by approximating the Hessian with a diagonal matrix with different elements on the diagonal. In other words, we have a vector $\boldsymbol{\eta}_t$ where $\text{diag}(\boldsymbol{\eta}_t) \approx \left(\mathbf{H}_{c(\mathbf{w}_t)}\right)^{-1}$. The resulting update is one that uses a vector of stepsizes, resulting in a slightly better approximation

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \boldsymbol{\eta}_t \cdot \nabla c(\mathbf{w}_t) \tag{4.3}$$

where $\cdot$ indicates elementwise multiplication.

The selection of this step-size is an important consideration. We discuss a few basic strategies to select the step-size in Section 4.4. Usually we assume the stepsize changes with each iteration t. It is likely that the Hessian is different at different points of the surface, since it reflects the local curvature. Just like a single scalar would be a poor approximation to $\left(\mathbf{H}_{c(\mathbf{w}_t)}\right)^{-1}$, it is a coarse approximation to assume the Hessian does not change for different $\mathbf{w}_t$. Most stepsize selection approaches attempt to approximate the local curvature, to some extent, and so use a vector stepsize that changes with time.

We can also reason about the role of a first-order versus second-order update once we use stochastic gradient descent (SGD). Recall that in SGD we use an (unbiased) sample estimate of the gradient $\nabla c(\mathbf{w}_t)$ with a mini-batch $\mathcal{B}_t$ of b indices subsampled from $\{1, 2, \ldots, n\}$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \tfrac{1}{b}\boldsymbol{\eta}_t \cdot \sum_{i \in \mathcal{B}_t} \nabla c_i(\mathbf{w}_t) \tag{4.4}$$

where $c(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \nabla c_i(\mathbf{w})$. We can interpret this update as a noisy variant of the batch GD update, because

$$\sum_{i\in\mathcal{B}_t} \nabla c_i(\mathbf{w}_t) = \nabla c(\mathbf{w}_t) + \underbrace{(\sum_{i\in\mathcal{B}_t} \nabla c_i(\mathbf{w}_t) - \nabla c(\mathbf{w}_t))}_{\epsilon} \tag{4.5}$$

where ϵ is a zero-mean random variable. On average, it is zero, so SGD on average behaves like GD. As long as we control the variance of the noise, then we still get nice convergence properties but with significantly less computation per step.

In fact, we can theoretically characterize the convergence rates for second-order GD, first-order GD and (first-order) SGD. We go into greater depth about this in Appendix B, and provide only a summary of the outcomes here. To discuss convergence rates, we need to decide on our convergence condition. Let us assume we have converged when the norm of the gradient is small: $\|\nabla c(\mathbf{w}_t)\| \leq \epsilon$ for some $\epsilon > 0$. Further, assume that our function is smooth, characterized by some constant L that reflects how quickly the gradient can change. Larger L imply that the function can change more quickly. Then we can show that GD converges with

$$\text{gradient norm} \leq 2L\frac{1}{t} \qquad \underbrace{(c(\mathbf{w}_0) - c(\mathbf{w}^*))}_{\text{distance of objective value at initialization to optimal}}$$

If we want the right-hand side to be less than ϵ, then we need $t \geq 2L\frac{1}{\epsilon}(c(\mathbf{w}_0) - c(\mathbf{w}^*))$. We write that the convergence rate is $t = O(1/\epsilon)$ since the other terms are problem-specific constants. Therefore, since each update costs $O(nd)$ and we have $O(1/\epsilon)$ iterations, then GD converges with $O(nd/\epsilon)$ computation. This result is true even for nonconvex functions, since the result is about converging to a stationary point, rather than to a global minima.

Now we can ask if this convergence rate is good. Notice that converging with $\epsilon = 0.1$ only requires $t = 10$, whereas getting higher and higher precision costs a lot: $\epsilon = 0.01$ requires $t = 100$, $\epsilon = 0.001$ requires $t = 1000$ and $\epsilon = 0.0001$ requires $t = 10000$. This is an exponential growth for increased precision. We can also write that this convergence rate corresponds to having $c(\mathbf{w}_t) - c(\mathbf{w}_\infty) = O(1/t)$, where $\mathbf{w}_\infty$ is the stationary point we eventually converge to. This rate is called *sublinear* because it slows down: the longer you run the algorithm, the less progress is makes.

Second-order gradient descent can obtain a linear convergence rate. It takes only $t = O(\log(1/\epsilon))$ iterations to converge. If we want $\epsilon = 0.1$ then we have $t = 1$, for $\epsilon = 0.01$ we have $t = 2$, for $\epsilon = 0.001$ we have $t = 3$ and $\epsilon = 0.0001$ requires $t = 4$. That is a massive difference to first-order GD. But, each update is much more expensive. Recall each update cost $O(nd^2)$. Using these formulas, we could decide whether to use second-order or first-order based on the number of samples, input dimensionality and desired precision.

Exercise 20: Consider a variety of choices for n, d and ϵ for $n > d$. We know GD needs $O(nd/\epsilon)$ to converge and second-order GD needs $O(nd^2\log(1/\epsilon))$ to converge. Which cases would you prefer one or the other? You can also consider cases when $n < d$, but then second-order GD uses $O(d^3\log(1/\epsilon))$. □

For SGD, the convergence rate depends on how we select the stepsize. If we use a constant stepsize, then we can actually get just as fast convergence as GD! But, with a

small catch. We will converge quickly to a neighborhood around the stationary point, but then oscillate with error proportional to the variance of the noise and the size of the stepsize. So our convergence is sublinear to this base level of error: $O(1/t)+$error-term. If we want to converge to the stationary point, then we can reduce the stepsize over time. For example, if we reduce the stepsize as $1/\sqrt{t}$, then we get a convergence rate of $O(1/\sqrt{t})$. This is already unfortunately slower. If we are too aggressive in the decrease, say decreasing as $1/t$, then the rate is $O(1/\log(t))$, which is very slow.

But, the convergence rates for SGD are mostly a good-news story. We can get convergence rates as fast as GD as long as we are careful about controlling the error due to noise. For example, we could use a non-decreasing stepsize to converge quickly to a ball around the stationary point, and then start decaying the stepsize. Or, we could increase the size of the mini-batch after an initial phase to start reducing this residual error term, which is zero if the variance of the mini-batch is zero. In practice, we can largely get this reasonably fast $O(1/t)$ convergence but with much less compute per step.

You might also be wondering if we can improve the convergence rate of SGD using a second-order update. The answer unfortunately seems to be negative. However, the vector-stepsize algorithms we discuss in the next section, inspired by approximating a second-order update, do end up being useful for SGD as well.

4.4 Stepsize Selection and Momentum

Because selecting the step-size is such an important part of an effective descent algorithm, there are many ways to do so. In addition to line search, one of the most popular methods is to use quasi-second-order (or quasi-Newton) methods. As we saw, the inverse of the Hessian provides a good way to select the stepsize, but is typically too expensive to compute let alone invert. Quasi-second-order methods approximate the Hessian, with as little storage and computation as possible. One of the simplest such approximations is to approximate only the diagonal of the Hessian, and then invert it, which only costs $O(d)$ computation and space. Such an approximation is typically quite poor for even the diagonal of the inverse Hessian, and so is not commonly used. Instead, the heuristic algorithms below seem to provide a better alternative.[1].

Our key question is how to get an effective vector stepsize. The idea is that you might need to take a bigger step in one dimension and a smaller in another dimension, due to curvature differences. For example, if in one direction, the optimization surface is flatter, you might need a bigger stepsize, and if another it is steep, then you need a small stepsize. One theoretically-motivated algorithm that implements this idea is *Adagrad*, which obtains vector $\boldsymbol{\eta}_t \in \mathbb{R}^{d+1}$ with

$$\boldsymbol{\eta}_t = (1 + \bar{\mathbf{g}}_t)^{-1/2} \tag{4.6}$$

where $\bar{\mathbf{g}}_t = \bar{\mathbf{g}}_{t-1} + \mathbf{g}_t^2$ using elementwise addition and powers. In other words, for each entry $\eta_{t,j}$ in the vector $\boldsymbol{\eta}_t$ and entry $\bar{g}_{t,j}$ in the vector $\bar{\mathbf{g}}_t$, we update $\bar{g}_{t,j} = \bar{g}_{t-1,j} + g_{t,j}^2$ and $\eta_{t,j} = (1+\bar{g}_{t,j})^{-1/2}$. Then each entry in the weights is updated using $w_{t+1,j} = w_{t,j} - \eta_{t,j} g_{t,j}$. This is the stepsize approach we use in Algorithm 1.

[1]Note that one of the most popular methods for GD has been LBFGS [17], which attempts to find a low rank approximation to the Hessian that is efficient to compute with reasonable storage. But, this algorithm is not easy to extend to SGD and has largely lost favor

Algorithm 1: SGD for objective $c(\mathbf{w}) = \frac{1}{n}c_i(\mathbf{w})$ with AdaGrad

Fix iteration parameters: number of epochs $= 10^4$ and mini-batch size $b = 32$
$\mathbf{w} \leftarrow$ random vector in $\mathbb{R}^d$
$\bar{\mathbf{g}} \leftarrow$ zero vector in $\mathbb{R}^d$
for $p = 1, \ldots$ number of epochs **do**
  Shuffle ordering of data points from $1, \ldots, n$
  for $k = 0, \ldots, \lfloor \frac{n}{b} \rfloor$ **do**
    $\mathbf{g} \leftarrow 0$
    $c \leftarrow 0$
    for $i = kb, \ldots, \min((k+1)b - 1, n)$ **do**
      $\mathbf{g} \leftarrow \mathbf{g} + \nabla c_i(\mathbf{w})$ ▷ for linear regression, $\nabla c_i(\mathbf{w}) = (\mathbf{x}_i^\top \mathbf{w} - y_i)\mathbf{x}_i$
      $c \leftarrow c + 1$
    $\mathbf{g} \leftarrow \mathbf{g}/c$ ▷ element-wise division
    for $j = 0, \ldots, d - 1$ **do**
      $\bar{\mathbf{g}}[j] \leftarrow \bar{\mathbf{g}}[j] + \mathbf{g}[j]^2$
      $\eta \leftarrow 1/(\sqrt{\bar{\mathbf{g}}[j] + 1})$
      $\mathbf{w}[j] \leftarrow \mathbf{w}[j] - \eta \mathbf{g}[j]$
return $\mathbf{w}$

In addition to Adagrad, there are a variety of other stepsize selection strategies. A few examples include Adadelta [32] and RMSProp, with a more comprehensive list given in a recent empirical study comparing methods [24]. As yet it is not clear that any one method has clear dominance over any others.

A common default is the Adam algorithm [14]. This algorithm essentially combines RMSProp and *momentum*. The idea behind momentum is to take bigger steps when the gradient direction has been the same recently, and take smaller steps if the direction changes. For this reason, it is also called the *heavy ball* method, since it prefers to keep rolling if it has been pushed multiple times in the same direction, and dampens movement if the two directions are different.[2]

The update equations remain simple. SGD with a fixed stepsize and momentum has update equations

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta \mathbf{g}_t + \beta(\mathbf{w}_t - \mathbf{w}_{t-1})$$

We can rewrite this to maintain an explicit momentum vector $\eta \mathbf{m}_t = \mathbf{w}_t - \mathbf{w}_{t-1}$. To see why, let us start by unrolling the recursion

$$\begin{aligned}
\beta(\mathbf{w}_t - \mathbf{w}_{t-1}) &= -\beta\eta \mathbf{g}_{t-1} + \beta^2(\mathbf{w}_{t-1} - \mathbf{w}_{t-2}) \\
&= -\beta\eta \mathbf{g}_{t-1} - \beta^2 \eta \mathbf{g}_{t-2} + \beta^3(\mathbf{w}_{t-2} - \mathbf{w}_{t-3})) \\
&= \ldots = -\eta \left(\beta \mathbf{g}_{t-1} + \beta^2 \mathbf{g}_{t-2} + \ldots + \beta^t \mathbf{g}_0 \right)
\end{aligned}$$

Therefore, we can update recursively update the momentum vector to be the sum of these

[2] Looking back at convergence rates, it has been shown that using the heavy ball method improves the convergence rate of GD from $O(1/t)$ to $O(1/t^2)$.

gradient vectors, exponentially scaled by $\beta < 1$, to rewrite the SGD update with momentum

$$\begin{aligned}\mathbf{w}_{t+1} &\leftarrow \mathbf{w}_t - \eta \mathbf{m}_{t+1} \\ \mathbf{m}_{t+1} &\leftarrow \mathbf{g}_t + \beta \mathbf{m}_t.\end{aligned}$$

Adam combines an RMSProp-like vector stepsize with momentum, with some modifications. First, it explicitly uses an exponential average for the momentum, by incorporating normalization by $1 - \beta$. In other words, it uses $(1 - \beta)\mathbf{m}_t$ instead of $\mathbf{m}_t$. Second, it uses a bias-correction to account for the fact that, early in the optimization, the sum of gradients in the momentum term would be skewed by the initialization of $\mathbf{m}_0 = \mathbf{0}$. Finally, it uses a vector of stepsizes that use a normalization by an exponential average of the squared values of the gradients. It maintains another vector, $\mathbf{v}_t$ for this exponential average with its own exponential weighting β_v. The resulting update equations are

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow (1 - \beta)\mathbf{g}_t + \beta \mathbf{m}_t \\ \mathbf{v}_{t+1} &\leftarrow (1 - \beta_v)\mathbf{g}_t^2 + \beta_v \mathbf{v}_t \\ \tilde{\mathbf{m}}_{t+1} &\leftarrow \mathbf{m}_{t+1}/(1 - \beta^t) && \triangleright \text{ bias correction} \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \mathbf{v}_{t+1}/(1 - \beta_v^t) && \triangleright \text{ bias correction} \\ \mathbf{w}_{t+1} &\leftarrow \mathbf{w}_t - \eta \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}\end{aligned}$$

for a small ϵ to avoid dividing by zero. Typical default values are $\beta = 0.9$, $\beta_v = 0.999$ and $\epsilon = 10^{-8}$. The stepsize likely needs to be tuned for the problem, but a common starting point is quite small, around $\eta = 0.001$.

Chapter 5

Generalized Linear Models

For prediction, you have seen linear regression and logistic regression. These are both actually instances of a more general class of predictors called generalized linear models (GLMs). Intuitively, GLMs extend ordinary least-squares regression beyond Gaussian probability distributions and beyond linear dependencies between the features and the target. Formally, they allow for $p(y|\mathbf{x})$ to be any *natural exponential family model*, of which the Gaussian (linear regression) and Bernoulli (logistic regression) are special cases.

We shall first revisit the formalization for ordinary least-squares regression. There, we assumed that a set of i.i.d. data points with their targets $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ were drawn according to some distribution $p(\mathbf{x}, y)$. We also assumed that an underlying relationship between the features and the target was linear, i.e.

$$Y = \sum_{j=1}^{d} X_j \omega_j + \varepsilon,$$

where $\boldsymbol{\omega}$ was a set of unknown weights and ε was a zero-mean normally distributed random variable with variance σ^2. We will now slightly reformulate this model. In particular, it will be useful to separate the underlying linear relationship between the features and the target from the fact that Y was normally distributed. That is, we write that

1. $p(y|\mathbf{x}) = \mathcal{N}(\mu(\mathbf{x}), \sigma^2)$ (i.e., where the mean is a function of $\mathbf{x}$ and the variance is constant across $\mathbf{x}$)
2. $\mathbb{E}[Y|\mathbf{x}] = \mathbf{x}\boldsymbol{\omega}$

This way of formulating linear regression allows us (i) to generalize the framework to non-linear relationships between the features and the target as well as (ii) to use distributions other than the Gaussian.

We start first with an example for the Poisson distribution, and then introduce GLMs more generally. Finally, we use this general class to derive multinomial logistic regression, which generalizes logistic regression from binary classification to multi-class classification. The goal of this chapter is both to introduce you to GLMs and multinomial logistic regression, as well as revisit MLE for prediction now for a broader set of models.

5.1 A First Example: The Poisson Distribution

We will start first with an example of a GLM, before moving on to the general class and general definition. Assume that data points correspond to cities in the world—described by some numerical features—and that the target variable is the number of sunny days

observed in a particular year. The target variable y may look like a Poisson distribution, given features $\mathbf{x}$. It would be more natural, therefore, to model

$$p(y|\mathbf{x}) = \text{Poisson}(\lambda) = \frac{\lambda^y \exp(-\lambda)}{y!}$$

where $\lambda > 0$ is the parameter (mean) of the Poisson distribution: $\mathbb{E}[Y|\mathbf{x}] = \lambda$. However, because $\lambda \in \mathbb{R}^+$, it would not be appropriate to model λ with $\mathbf{x}\boldsymbol{\omega} \in \mathbb{R}$. Rather, we would like to transfer our linear prediction with some function g to adjusts the range of the linear combination of features to the domain of the parameters of the probability distribution.

We can do so by introducing an *exponential transfer* for this Poisson distribution, and more generically, later any invertible transfer function g. If we can instead estimate $\boldsymbol{\omega}$ such that $\lambda = \exp(\mathbf{x}\boldsymbol{\omega})$, then we can guarantee our estimates are in the correct range. Alternatively, one can consider that we are learning a linear weighting of features to learn a transformed parameter, $\log(\lambda) = \mathbf{x}\boldsymbol{\omega}$. This simple modification is why these models are called *generalized* linear models, because the key component is still a linear weighting. We formalize the types of distributions and transfers that can be considered in the below sections, but first finish off this example with Poisson regression to provide a concrete example.

To establish the GLM model for Poisson regression, we assume (1) an exponential transfer between the expectation of the target and linear combination of features, and (2) the Poisson distribution for the target variable.

1. $p(y|\mathbf{x}) = \text{Poisson}(\lambda(\mathbf{x}))$, where $\lambda(\mathbf{x}) = E[Y|\mathbf{x}]$
2. $\mathbb{E}[Y|\mathbf{x}] = \exp(\mathbf{x}\boldsymbol{\omega})$ or $\log(\mathbb{E}[Y|\mathbf{x}]) = \mathbf{x}\boldsymbol{\omega}$

The resulting probability distribution, for $y \in \mathbb{N}$, is

$$p(y|\mathbf{x}, \boldsymbol{\omega}) = \frac{\exp(\mathbf{x}\boldsymbol{\omega} y) \exp(-\exp(\mathbf{x}\boldsymbol{\omega}))}{y!}$$

where $\lambda^y = \exp(\mathbf{x}\boldsymbol{\omega})^y = \exp(\mathbf{x}\boldsymbol{\omega} y)$ because $\exp(a)^b = \exp(ab)$.

We can use maximum likelihood estimation to find the parameters of this regression model. Our objective $c(\mathbf{w})$ corresponds to the negative log-likelihood

$$c(\mathbf{w}) = \tfrac{1}{n}\sum_{i=1}^{n} c_i(\mathbf{w}) = \tfrac{1}{n}\sum_{i=1}^{n} -\ln p(y_i|\mathbf{x}_i, \mathbf{w})$$
$$\text{where } c_i(\mathbf{w}) \stackrel{\text{def}}{=} -\ln p(y_i|\mathbf{x}_i, \mathbf{w}) = -\mathbf{x}_i\mathbf{w}y_i + \exp(\mathbf{x}_i\mathbf{w}) + \ln y_i!$$

To minimize $c(\mathbf{w})$, we want to find a stationary point, namely find $\mathbf{w}_0$ such that $\nabla c(\mathbf{w}_0) = \mathbf{0}$. This formula, however, does not have a closed-form solution. Therefore, unlike linear regression, we will have to use gradient descent. We could choose to use first-order or second-order gradient descent, and batch or stochastic gradient descent.

The key step in any of these is to first compute the gradient for one sample. We start by deriving the partial derivative of the negative log-likelihood for one sample

$$\begin{aligned}\frac{\partial c_i(\mathbf{w})}{\partial w_j} &= \exp(\mathbf{x}_i\mathbf{w})x_{ij} - x_{ij}y_i \\ &= x_{ij}\left(\exp(\mathbf{x}_i\mathbf{w}) - y_i\right) \\ &= x_{ij} \cdot \left(p_i - y_i\right).\end{aligned}$$

where $p_i \stackrel{\text{def}}{=} \exp(\mathbf{x}_i\mathbf{w})$ is the prediction and $p_i - y_i$ corresponds to a prediction error for sample i. The batch gradient is

$$\begin{aligned}\nabla c(\mathbf{w}) &= \tfrac{1}{n}\sum_{i=1}^{n}\nabla c_i(\mathbf{w}) = \tfrac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i^\top\,(p_i - y_i) \\ &= \tfrac{1}{n}\mathbf{X}^\top\,(\mathbf{p}-\mathbf{y})\end{aligned} \tag{5.1}$$

where $\mathbf{p}$ is a vector with elements $p_i = \exp(\mathbf{x}_i\mathbf{w})$, and $\mathbf{p}-\mathbf{y}$ is an error vector. For stochastic gradient descent, each step consists of using the gradient for one sample (i.e., $\nabla c_i(\mathbf{w}_t)$) and for batch gradient descent, each step consists of using the gradient for all samples (i.e., $\nabla c(\mathbf{w}_t)$).

We can additionally consider the Hessian matrix, both to evaluate the properties of the stationary points as well as to allow for second-order gradient descent—though it is likely too expensive if d is large. The second partial derivative of the negative log likelihood function for one sample is

$$\begin{aligned}\frac{\partial^2 c_i(\mathbf{w})}{\partial w_j \partial w_k} &= x_{ij}\exp(\mathbf{x}_i\mathbf{w})x_{ik} \\ &= x_{ij}p_i x_{ik} \\ \text{with}\quad \frac{\partial^2 c(\mathbf{w})}{\partial w_j \partial w_k} &= \sum_{i=1}^{n}\frac{\partial^2 c_i(\mathbf{w})}{\partial w_j \partial w_k} = \sum_{i=1}^{n} x_{ij}p_i x_{ik}.\end{aligned}$$

For $\mathbf{P}$ an $n \times n$ diagonal matrix with p_i on the diagonal, the Hessian matrix is therefore

$$H_{c(\mathbf{w})} = \tfrac{1}{n}\mathbf{X}^\top\mathbf{P}\mathbf{X}. \tag{5.2}$$

This matrix is positive definite if $\mathbf{X}$ is not low-rank, which would mean there is only one stationary point and that it is the global minimum. In fact, we know that the objective for Poisson regression is convex, even if $\mathbf{X}$ is not full rank, and so all stationary points are global minima. If $\mathbf{X}$ is not full rank, then there is a space of many equivalent solutions (infinitely many). This is because many $\mathbf{w}$ produce the same $\mathbf{X}\mathbf{w}$, and so the same predictions. If $H_{c(\mathbf{w})}$ is in-fact semi-definite, reflecting that we have a flat part in the curve of equal loss for all of these solutions.

Exercise 21: What is the second-order update for Poisson regression? □

5.2 Exponential Family Distributions

In the previous section, we used a specific example to illustrate how to generalize beyond Gaussian distributions. The approach more generally extends to any exponential family distribution. We focus on the natural exponential family, which is sufficient for most generalized linear models. The natural exponential family is a class of probability distributions with the following form

$$p(y|\theta) = \exp\left(\theta y - a(\theta) + b(y)\right)$$

where $\theta \in \mathbb{R}$ is the parameter to the distribution, $a : \mathbb{R} \to \mathbb{R}$ is a log-normalizer function and $b : \mathbb{R} \to \mathbb{R}$ is a function of only y that will typically be ignored in our optimization

because it is not a function of θ. Many of the often encountered (families of) distributions are members of the exponential family; e.g. exponential, Gaussian, Gamma, Poisson, or the binomial distributions. Therefore, it is useful to generically study the exponential family to obtain algorithms for each these distributions.

Example 8: The Poisson distribution can be expressed as

$$p(y|\lambda) = \exp\left(y \log \lambda - \lambda - \log y!\right),$$

where $\lambda \in \mathbb{R}^+$ and $\mathcal{Y} = \mathbb{N}_0$. Thus, $\theta = \log \lambda$, $a(\theta) = \exp(\theta)$, and $b(y) = -\log y!$. □

Now let us get some further insight into the properties of the exponential family parameters and why this class is convenient for estimation. The function $a(\theta)$ is typically called the log-partitioning function or simply a log-normalizer. It is called this because

$$a(\theta) = \log \int_{\mathcal{Y}} \exp\left(\theta y + b(y)\right) dy$$

and so plays the role of ensuring that we have a valid density: $\int_{\mathcal{Y}} p(y)dy = 1$. (For discrete Y, this integral is a sum.) Importantly, for many common GLMs, the derivative of a corresponds to the transfer function. For example, for Poisson regression, the transfer function is $g(\theta) = \exp(\theta)$, and the derivative of a is $\exp(\theta)$. Therefore, the log-normalizer for an exponential family informs what transfer g should be used. This result is not too surprising, given the fact that we can show that

$$\frac{\partial a(\theta)}{\partial \theta} = \mathbb{E}\left[Y\right] \qquad \text{and} \qquad \frac{\partial^2 a(\theta)}{\partial \theta^2} = \text{Var}\left[Y\right]$$

and $g(\theta)$ is exactly modelling $\mathbb{E}\left[Y\right]$.

5.3 Formalizing Generalized Linear Models

We shall now formalize GLMs. The two key components of GLMs can be expressed as

1. $p(y|\mathbf{x})$ is an Exponential Family distribution with log-normalizer a.
2. $\mathbb{E}[Y|\mathbf{x}] = g(\mathbf{x}\boldsymbol{\omega})$ or $g^{-1}(\mathbb{E}[y|\mathbf{x}]) = \mathbf{x}\boldsymbol{\omega}$ where $g(\theta) = \frac{\partial a(\theta)}{\partial \theta}$.

The function g is called the *transfer* function.[1] For Poisson regression, g is the exponential function, and as we shall see for logistic regression, g is the sigmoid function. The transfer function adjusts the range of $\mathbf{x}\boldsymbol{\omega}$ to the domain of Y; because of this relationship, transfer functions are usually not selected independently of the distribution for Y. The generalization to the exponential family from the Gaussian distribution allows us to model a much wider range of target functions.

To relate these more clearly to exponential family distributions, we have to consider conditional distributions. Each $p(y|\mathbf{x})$ is an exponential family distribution, with parameter

[1] The inverse of g is called the link function, but, really, we don't need to name everything. For simplicity in terminology, we only ever refer to the transfer or the inverse of the transfer.

$\theta = \mathbf{x}\mathbf{w}$. When learning $\mathbf{w}$—by maximizing likelihood—we are learning the parameter θ_i for each sample $(\mathbf{x}_i, y_i)$. The negative log-likelihood for one sample is

$$\begin{aligned} c_i(\mathbf{w}) &= -\ln \exp(\theta_i y_i - a(\theta_i) + b(y_i)) \\ &= -\theta_i y_i + a(\theta_i) - b(y_i) \\ &= -(\mathbf{x}_i\mathbf{w})y_i + a(\mathbf{x}_i\mathbf{w}) - b(y_i) \\ &= a(\mathbf{x}_i\mathbf{w}) - (\mathbf{x}_i\mathbf{w})y_i - b(y_i) \end{aligned}$$

with gradients

$$\begin{aligned} \frac{\partial c_i(\mathbf{w})}{\partial w_j} &= \frac{\partial a(\theta_i)}{\partial w_j} - \frac{\partial \theta_i}{\partial w_j} y_i \\ &= \frac{\partial a(\theta_i)}{\partial \theta_i}\frac{\partial \theta_i}{\partial w_j} - \frac{\partial \theta_i}{\partial w_j} y_i \\ &= \left(\frac{\partial a(\theta_i)}{\partial \theta_i} - y_i\right)\frac{\partial \theta_i}{\partial w_j} \\ &= \left(\frac{\partial a(\theta_i)}{\partial \theta_i} - y_i\right) x_{ij} \\ &= (g(\theta_i) - y_i)\, x_{ij} \end{aligned}$$

where the last step follows from the fact that above we mentioned that the transfer function is chosen such that $g(\theta) = a'(\theta)$, the derivative of a. Therefore, for every natural exponential family distribution, we have a clear likelihood objective and gradient to estimate the parameters for the conditional distribution $p(y|\mathbf{x})$.

Let us now consider what the gradient updates look like. Given the appropriate transfer g for the desired exponential family distribution, the stochastic gradient descent update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \left(g(\mathbf{x}_i\mathbf{w}_t) - y_i\right)\mathbf{x}_i^\top$$

and the batch gradient descent update is

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{\eta_t}{n}\sum_{i=1}^{n}\left(g(\mathbf{x}_i\mathbf{w}_t) - y_i\right)\mathbf{x}_i^\top \\ &= \mathbf{w}_t - \frac{\eta_t}{n}\mathbf{X}^\top\left(\mathbf{p} - \mathbf{y}\right) \end{aligned}$$

where $\mathbf{p} \overset{\text{def}}{=} [p_1, \ldots, p_n] \in \mathbb{R}^{n\times 1}$ is a column vector for $p_i \overset{\text{def}}{=} g(\mathbf{x}_i\mathbf{w}_t)$. The second partial derivative of the negative log likelihood function for one sample and across samples, respectively, is

$$\begin{aligned} \frac{\partial^2 c_i(\mathbf{w})}{\partial w_j \partial w_k} &= x_{ij}\frac{\partial g(\theta_i)}{\partial \theta_i}x_{ik} = x_{ij}g'(\mathbf{x}_i^\top\mathbf{w})x_{ik} \\ \frac{\partial^2 c(\mathbf{w})}{\partial w_j \partial w_k} &= \tfrac{1}{n}\sum_{i=1}^{n}\frac{\partial^2 c_i(\mathbf{w})}{\partial w_j \partial w_k} = \tfrac{1}{n}\mathbf{X}_{:j}^\top\mathbf{D}\mathbf{X}_{:k} \end{aligned}$$

for $\mathbf{D}$ an $n \times n$ diagonal matrix with $\frac{\partial g(\theta_i)}{\partial \theta_i}$ on the diagonal. This last line follows from the fact that this diagonal matrix weights each product $x_{ij}x_{ik}$ with the corresponding diagonal

entry. The term $\mathbf{X}_{:j}^\top \mathbf{D}\mathbf{X}_{:k}$ gives a weighted dot product, where the vectors are size n and the dot product provides a sum over samples: $\mathbf{X}_{:j}^\top \mathbf{D}\mathbf{X}_{:k} = \sum_{i=1}^n \mathbf{D}_{ii} x_{ij} x_{ik}$. The Hessian matrix is therefore

$$H_{c(\mathbf{w})} = \begin{bmatrix} \frac{\partial^2 c}{\partial w_1^2}(\mathbf{w}_0) & \frac{\partial^2 c}{\partial w_1 \partial w_2}(\mathbf{w}_0) & \cdots & \frac{\partial^2 c}{\partial w_1 \partial w_d}(\mathbf{w}_0) \\ \frac{\partial^2 c}{\partial w_2 \partial w_1}(\mathbf{w}_0) & \frac{\partial^2 c}{\partial w_2^2}(\mathbf{w}_0) & & \\ \vdots & \vdots & \ddots & \\ \frac{\partial^2 c}{\partial w_d \partial w_1}(\mathbf{w}_0) & \cdots & & \frac{\partial^2 c}{\partial w_d^2}(\mathbf{w}_0) \end{bmatrix} = \frac{1}{n}\mathbf{X}^\top \mathbf{D}\mathbf{X}. \tag{5.3}$$

As in Poisson regression, this matrix is guaranteed to be positive semi-definite, and further positive definite if $\mathbf{X}$ is not low-rank. The second-order gradient descent update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - (\mathbf{X}^\top \mathbf{D}\mathbf{X})^{-1} \mathbf{X}^\top (\mathbf{p} - \mathbf{y})$$

Note that the chosen transfer does not necessarily have to correspond to the derivative of a. Rather, this provides a mechanism for ensuring a nice loss function (see Appendix A.3 for more on why). However, this does not mean that any other transfer will necessarily result in an undesirable loss function. Without any reason to prefer a different transfer, however, it is definitely sensible to stick with $g = a'$.

5.4 Revisiting Logistic Regression

One of the most popular uses of GLMs is a combination of a Bernoulli distribution with a sigmoid transfer function: logistic regression. We summarize the logistic regression model as follows

1. $p(y|\mathbf{x}) = \text{Bernoulli}(\alpha(\mathbf{x}))$ with $\alpha(\mathbf{x}) = \mathbb{E}[Y|\mathbf{x}]$
2. $\mathbb{E}[Y|\mathbf{x}] = \sigma(\mathbf{x}\boldsymbol{\omega})$ or $\text{logit}(\mathbb{E}[Y|\mathbf{x}]) = \mathbf{x}\boldsymbol{\omega}$

where $\text{logit}(x) = \ln \frac{x}{1-x}$, $y \in \{0, 1\}$, and $\alpha \in (0, 1)$ is the parameter (mean) of the Bernoulli distribution. Recall that the sigmoid function $\sigma : \mathbb{R} \to [0, 1]$ is defined as

$$\sigma(\theta) \stackrel{\text{def}}{=} \frac{1}{1 + e^{-\theta}}$$

where here we apply the sigmoid to $\theta = \mathbf{x}\boldsymbol{\omega}$. Using the generic formula above, the SGD update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \left(\sigma(\mathbf{x}_i \mathbf{w}_t) - y_i\right) \mathbf{x}_i^\top$$

Exercise 22: Derive the second-order gradient descent update for logistic regression. □

5.5 Multinomial Logistic Regression

Now let us consider discriminative multiclass classification, where $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{1, 2, \ldots, m\}$. This setting arises naturally in machine learning, where there is often more

than two categories. For example, if we want to predict the blood type (A, B, AB and O) of an individual, then we have four classes. Here we discuss multiclass classification where we only want to label a datapoint with one class out of m. In other settings, one might want to label a datapoint with multiple classes or labels; this is briefly discussed at the end of this section.

We can nicely generalize to this setting using the idea of multinomials and the corresponding transfer function, as with the other generalized linear models. To use this distribution, we will assume that the target class is an indicator vector. For example, if $m = 4$, and the class for an input $\mathbf{x}_i$ is 2, then $\mathbf{y}_i = [0\ 1\ 0\ 0]$. This is equivalent to writing $y_i = 2$, but using an indicator vector makes it more straightforward to use the multinomial distribution. The multinomial distribution is a member of the exponential family, and so we can leverage the same update rules for GLMs. We can write

$$\begin{aligned} p(\mathbf{y}|\mathbf{x}) &= \frac{1}{y_1! \dots y_m!} p(y_1 = 1|\mathbf{x})^{y_1} \dots, p(y_m = 1|\mathbf{x})^{y_m} \\ &= p(y_1 = 1|\mathbf{x})^{y_1} \dots, p(y_m = 1|\mathbf{x})^{y_m} \qquad \triangleright\ 0! = 1! = 1, \text{ each } y_k = 0 \text{ or } 1 \end{aligned} \tag{5.4}$$

where the usual numerator $n! = 1$ because $n = \sum_{j=1}^{m} y_j = 1$ since we can only have one class value.

As with logistic regression, we can parametrize $p(y_j = 1|\mathbf{x})$ using $\sigma(\mathbf{x}\mathbf{w}_j)$. However, we must also ensure that $\sum_{j=1}^{m} p(y_j = 1|\mathbf{x}) = 1$, which we do by normalizing each $\sigma(\mathbf{x}\mathbf{w}_j)$ by this sum. The *softmax* transfer for multinomial logistic regression does just that. The parameters can be represented as a matrix $\mathbf{W} \in \mathbb{R}^{d \times m}$ where $\mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_m]$ is composed of m weight vectors. The transfer for this setting is the softmax transfer

$$\begin{aligned} \text{softmax}(\mathbf{x}^\top \mathbf{W}) &= \left[\frac{\exp(\mathbf{x}\mathbf{w}_1)}{\sum_{j=1}^{m} \exp(\mathbf{x}\mathbf{w}_j)}, \dots, \frac{\exp(\mathbf{x}\mathbf{w}_m)}{\sum_{j=1}^{m} \exp(\mathbf{x}\mathbf{w}_j)} \right] \\ &= \left[\frac{\exp(\mathbf{x}\mathbf{w}_1)}{\exp(\mathbf{x}\mathbf{W})\mathbf{1}}, \dots, \frac{\exp(\mathbf{x}\mathbf{w}_m)}{\exp(\mathbf{x}\mathbf{W})\mathbf{1}} \right] \end{aligned}$$

and the prediction is $\text{softmax}(\mathbf{x}) = \hat{\mathbf{y}} \in [0, 1]^m$, which gives the probability in each entry of being labeled as that class, where $\hat{\mathbf{y}}^\top \mathbf{1} = 1$ signifying that the probabilities sum to 1.

With the parameters of the model parameterized by $\mathbf{W}$ and the softmax transfer, we can determine the maximum likelihood formulation. By plugging in the parameterization into Equation (5.4), taking the negative log of that likelihood and dropping constants, we arrive at the following loss for samples $(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_n, \mathbf{y}_n)$

$$\min_{\mathbf{W} \in \mathbb{R}^{d \times m}} \sum_{i=1}^{n} \log\left(\exp(\mathbf{x}_i \mathbf{W})\mathbf{1}\right) - \mathbf{x}_i \mathbf{W} \mathbf{y}_i^\top$$

where $\mathbf{y}_i$ is a row vector $(1 \times m)$, so multiplying $\mathbf{W}\mathbf{y}_i^\top$ results in a vector. The gradient is

$$\begin{aligned} \nabla \sum_{i=1}^{n} \left(\log\left(\exp(\mathbf{x}_i \mathbf{W})\mathbf{1}\right) - \mathbf{x}_i \mathbf{W} \mathbf{y}_i^\top \right) &= \sum_{i=1}^{n} \frac{\mathbf{x}_i^\top \exp(\mathbf{x}_i \mathbf{W})}{\exp(\mathbf{x}_i \mathbf{W})\mathbf{1}} - \mathbf{x}_i^\top \mathbf{y}_i \\ &= \sum_{i=1}^{n} \mathbf{x}_i^\top \left[\frac{\exp(\mathbf{x}_i \mathbf{W})}{\exp(\mathbf{x}_i \mathbf{W})\mathbf{1}} - \mathbf{y}_i \right] \\ &= \sum_{i=1}^{n} \mathbf{x}_i^\top \left[\mathbf{p}_i - \mathbf{y}_i \right]. \end{aligned}$$

where the prediction vector $\mathbf{p}_i \stackrel{\text{def}}{=} \frac{\exp(\mathbf{x}_i\mathbf{W})^\top}{\exp(\mathbf{x}_i\mathbf{W})\mathbf{1}} \in \mathbb{R}^m$ is a row vector of predicted probabilities for each class for sample i. Notice that this means that $\mathbf{x}_i^\top [\mathbf{p}_i - \mathbf{y}_i]$ is an outer product, between a $d \times 1$ vector and $1 \times m$ vector, resulting in a $d \times m$ matrix. As before, we do not have a closed form solution for this gradient, and will use iterative methods to solve for $\mathbf{W}$. For example, the stochastic gradient descent update is

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta_t \mathbf{x}_i^\top \left[\text{softmax}(\mathbf{x}_i\mathbf{W}_t) - \mathbf{y}_i\right].$$

We maintain our parameters as a matrix to make the update more interpretable. To better match it to our regular SGD updates, though, we want to think of $\mathbf{W}_t$ as a vector. We can always do so simply by linearizing it: we imagine we stack all the columns of $\mathbf{W}$. Then if we want to use a vector step-size, for example, it corresponds to a vector of size dm. Alternatively, we can also have η be of the same shape as $\mathbf{W}$—namely a matrix of size $d \times m$—and we still use an element-wise product with the gradient which is also of size $d \times m$. Either interpretation is equivalent, and it is simply a matter of which is simpler to implement in the language of your choice. Note that the weights for each class are not learned independently, because they each impact the other classes due to being in the normalization in the softmax.

The final prediction $\text{softmax}(\mathbf{x}\mathbf{W}) \in [0,1]^m$ gives the probabilities of being in a class. As with logistic regression, to pick one class, the highest probability value is chosen. For example, with $m = 4$, we might predict [0.1 0.2 0.6 0.1] and so decide to classify the point into class 3. Formally, we predict class

$$\underset{y\in\{1,\ldots,m\}}{\text{argmax}}\ p(y|\mathbf{x}) = \underset{k\in\{1,\ldots,m\}}{\text{argmax}} \frac{\exp(\mathbf{x}\mathbf{w}_k)}{\sum_{j=1}^m \exp(\mathbf{x}\mathbf{w}_j)} = \underset{k\in\{1,\ldots,m\}}{\text{argmax}}\ \mathbf{x}\mathbf{w}_k$$

Pivoting around one of the classes There is one other nuance for multinomial logistic regression, which is often ignored in practice but important to know. The above updates every vector $\mathbf{w}_k$ for $k \in \{1,\ldots,m\}$. However, this is not actually necessary. In fact, we only need to learn $\mathbf{w}_k$ for $k \in \{1,\ldots,m-1\}$, because the probability for the final class can be inferred using the first $m-1$: $p(y_m = 1|\mathbf{x}) = 1 - \sum_{j=1}^{m-1} p(y_j = 1|\mathbf{x})$. Learning $\mathbf{w}_m$ is unnecessary, and instead we can fix $\mathbf{w}_m = \mathbf{0}$. The remaining variables will "pivot" around this final class, and be able to represent the same probabilities as if we allowed $\mathbf{w}_m$ to be learned.

In fact, we did just this in logistic regression, namely when we had only two classes. Recall the binary setting for logistic regression, where we learned one weight vector $\mathbf{w} \in \mathbb{R}^d$ and we had $\sigma(\mathbf{x}\mathbf{w}) = (1+\exp(-\mathbf{x}\mathbf{w}))^{-1} = \frac{\exp(\mathbf{x}\mathbf{w})}{1+\exp(\mathbf{x}\mathbf{w})}$. We can contrast this with multinomial logistic regression, with pivoting. The weights for multinomial logistic regression with two classes are $\mathbf{W} = [\mathbf{0}, \mathbf{w}]$ giving

$$p(y = 1|\mathbf{x}) = \frac{\exp(\mathbf{x}\mathbf{w})}{\exp(\mathbf{x}\mathbf{W})\mathbf{1}} = \frac{\exp(\mathbf{x}\mathbf{w})}{\exp(\mathbf{x}^\top\mathbf{0}) + \exp(\mathbf{x}\mathbf{w})} = \frac{\exp(\mathbf{x}\mathbf{w})}{1 + \exp(\mathbf{x}\mathbf{w})} = \sigma(\mathbf{x}\mathbf{w}).$$

Similarly, for $m > 2$, by fixing one of the classes weights to zero, say $\mathbf{w}_m = \mathbf{0}$, the other weights $\mathbf{w}_1, \ldots, \mathbf{w}_{m-1}$ are learned to ensure that $p(y = m|\mathbf{x}) = \frac{\exp(\mathbf{x}\mathbf{w}_m)}{\exp(\mathbf{x}\mathbf{W})\mathbf{1}} = \frac{1}{1+\sum_{j=1}^{m-1}\exp(\mathbf{x}\mathbf{w}_j)}$ and that $\sum_{j=1}^m p(y = j|\mathbf{x}) = 1$.

The optimization itself can be written slightly differently now.

$$\min_{\mathbf{W}\in\mathbb{R}^{d\times m}:\mathbf{W}_{:m}=\mathbf{0}} \sum_{i=1}^{n} \log\left(\exp(\mathbf{x}_i\mathbf{W})\mathbf{1}\right) - \mathbf{x}_i\mathbf{W}\mathbf{y}_i^\top$$

We now have a constraint on part of the variable. However, this was solely written this way for convenience. We do not optimize $\mathbf{W}_{:m}$, as it is fixed at zero; one can rewrite this minimization and gradient to only apply to the $\mathbf{W}_{:(1:m-1)}$. This corresponds to initializing $\mathbf{W}_{:m} = \mathbf{0}$, and then only using the first $m-1$ columns of the gradient in the update to $\mathbf{W}_{:(1:m-1)}$. Alternatively, you can also update all of $\mathbf{W}$ and then set the last weights to zero after every update:

$$\begin{aligned}\mathbf{W}_{t+1} &\leftarrow \mathbf{W}_t - \eta_t \mathbf{x}_i^\top \left[\text{softmax}(\mathbf{x}_i\mathbf{W}_t) - \mathbf{y}_i\right] \\ \mathbf{W}_{t+1,m} &\leftarrow \mathbf{0}\end{aligned}$$

This optimization that is only over $\mathbf{W}_{:(1:m-1)}$ is over fewer variables, and is likely to have a unique solution. This is in contrast to the update without the pivot, which has too many free variables and infinitely many equivalent solutions.

There are multiple benefits to pivoting. First, as just discussed, without the pivot, the solution to our optimization is not unique. There is an infinite space of equivalent solutions. When possible, forcing our solution to be unique is preferable. A related point is that to be a valid GLM, the transfer function must be invertible. Here, the softmax is only guaranteed to be invertible if we constrain (fix) the final variable, say to be 0. For example, imagine we compute the softmax of $[\theta_1, \theta_2, \ldots, \theta_{m-1}, 0]$ to get output $[y_1, y_2, \ldots, y_m]$. Then we can infer $\theta_1, \theta_2, \ldots, \theta_{m-1}$ from $[y_1, y_2, \ldots, y_m]$. However, if we used $[\theta_1, \theta_2, \ldots, \theta_{m-1}, \theta_m]$ to compute $\mathbf{y} = [y_1, y_2, \ldots, y_m]$, then there are infinitely many vectors $[\theta_1, \theta_2, \ldots, \theta_{m-1}, \theta_m]$ that could have produced the same $\mathbf{y}$. Therefore, we cannot undo this operation—the application of the softmax—to figure out what these variables were.

Second, when possible, it is better to learn fewer parameters. The ramifications of learning this additional vector are not actually clear—no such studies are available. But, erring on the side of simplicity and reducing the number of free variables is typically a good choice. Finally, as mentioned above, if we do pivot, then we can ensure that a special case of multinomial logistic regression for two classes reduces to same algorithm that we designed for the two class case: logistic regression.

Contrast to Multi-label Classification: In multi-label classification, the goal is to assign one or more labels to an item. Multinomial logistic regression is for multi-class classification, where each item is assigned to exactly one label. If you want to predict multiple labels for a datapoint $\mathbf{x}$, then a common strategy is to learn separate binary predictors for each label. Each predictor is queried separately, and a datapoint will label each class as 0 or 1, with potentially more than one class having a 1. Above, we examined the case where the datapoint was exclusively in one of the provided classes, by setting $n = 1$ in the multinomial.

Chapter 6

Constrained Optimization with Proximal Methods

In many cases, we will have constraints on our optimization. For example, when we considered parameter estimation for the Poisson distribution, we needed to ensure we found a parameter $\lambda > 0$. Our strategy involved simply finding the single stationary point, ensuring it was a local minimum and satisfied the constraint. Since the single stationary point is a local minimum, we know it is actually a global minimum and that the solution is not on the boundary of the constraint set, namely at 0.

More generally, however, this strategy may fail to find a valid solution in the constraint set. In this chapter, we discuss a general strategy to handle constraints using proximal methods, and then go through a use-case: ℓ_1 regularization for feature selection. This case study is for the case when we have a closed-form proximal operator, but that is not always the case. We then discuss what to do when we do not have a closed form.

6.1 Proximal Methods

We can generalize gradient descent using proximal methods that break up the optimization into two steps: a gradient descent step followed by a projection step. For example, imagine we wanted to do linear regression under the constraint that the weight $w_j \in [-1, 1]$. Recall that the gradient for linear regression at weights $\mathbf{w}_t$ was $\mathbf{g}_t \stackrel{\text{def}}{=} \sum_{i=1}^n (\mathbf{x}_i^\top \mathbf{w}_t - y_t)\mathbf{x}_i$. Then the two steps include

$$\tilde{\mathbf{w}}_{t+1} = \mathbf{w}_t - \eta_t \mathbf{g}_t$$

$$w_{t+1,j} = \begin{cases} -1 & \text{if } \tilde{w}_{t+1,j} < -1 \\ \tilde{w}_{t+1,j} & \text{if } \tilde{w}_{t+1,j} \in [-1, 1] \\ 1 & \text{if } \tilde{w}_{t+1,j} > 1 \end{cases}$$

If the gradient descent step takes you outside the constraint set, the second step projects back to that set. For the constraint $w_j \in [-1, 1]$, the projection is simply: any elements less than -1 are projected to -1, and any elements greater than 1 are projected to 1.

Proximal methods allow us to formalize this simple idea.[1] To write this generically, assume we have an optimization problem of the form

$$\min_{\mathbf{w} \in \mathbb{R}^d} c(\mathbf{w}) + r(\mathbf{w})$$

where we assume c is differentiable everywhere and r can be any nonsmooth function. We can encode our constraints using r. The smooth function c might be the least-squares loss

[1] For a much more thorough treatment of this topic, see [19].

and, for constraint set $\mathcal{F} = [-1, 1]^d$, we can set

$$r(\mathbf{w}) = \begin{cases} 0 & \text{if } \mathbf{w} \in \mathcal{F} \\ \infty & \text{otherwise} \end{cases}$$

To obtain the minimum of $c(\mathbf{w}) + r(\mathbf{w})$, we must find $\mathbf{w} \in \mathcal{F}$ as otherwise the objective is ∞ (which is clearly not minimal).

The proximal update is derived similarly to gradient descent, by using the Taylor series expansion. Recall that we found the first-order gradient descent update using

$$\mathbf{w}_{t+1} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} c(\mathbf{w}_t) + \nabla c(\mathbf{w}_t)^\top (\mathbf{w} - \mathbf{w}_t) + \frac{1}{2\eta_t} \|\mathbf{w} - \mathbf{w}_t\|_2^2$$

where the last term was the first-order approximation using a scalar stepsize, rather than the Hessian. Finding this minimum results in the update

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla c(\mathbf{w}_t)$$

Similarly, we can write the proximal update using the same expansion on c, in addition to including r

$$\begin{aligned}
\mathbf{w}_{t+1} &= \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} c(\mathbf{w}_t) + \nabla c(\mathbf{w}_t)^\top (\mathbf{w} - \mathbf{w}_t) + \frac{1}{2\eta_t} \|\mathbf{w} - \mathbf{w}_t\|_2^2 + r(\mathbf{w}) \\
&= \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \nabla c(\mathbf{w}_t)^\top (\mathbf{w} - \mathbf{w}_t) + \frac{1}{2\eta_t} \|\mathbf{w} - \mathbf{w}_t\|_2^2 + r(\mathbf{w}) && \triangleright \text{ dropped constant} \\
&= \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \eta_t \nabla c(\mathbf{w}_t)^\top (\mathbf{w} - \mathbf{w}_t) + \tfrac{1}{2} \|\mathbf{w} - \mathbf{w}_t\|_2^2 + \eta_t r(\mathbf{w}) && \triangleright \text{ multiply by } \eta_t \\
&= \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \tfrac{1}{2} \|\mathbf{w} - (\mathbf{w}_t - \eta_t \nabla c(\mathbf{w}_t))\|_2^2 + \eta_t r(\mathbf{w}) && \triangleright \text{ dropped constant}
\end{aligned}$$

where the last equality follows because the two equations are the same up to a constant: the term $\eta_t^2 \mathbf{g}_t^\top \mathbf{g}_t$ for $\mathbf{g}_t \stackrel{\text{def}}{=} \nabla c(\mathbf{w}_t)$ (see Appendix A.4.1 if you would like to see the steps).

We can rewrite this final form in two steps to more easily see the descent and project components. The proximal gradient updates take the form

$$\begin{aligned}
\tilde{\mathbf{w}}_{t+1} &= \mathbf{w}_t - \eta_t \mathbf{g}_t \\
\mathbf{w}_{t+1} &= \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \tfrac{1}{2} \|\mathbf{w} - \tilde{\mathbf{w}}_{t+1}\|_2^2 + \eta_t r(\mathbf{w})
\end{aligned}$$

where the second step can be seen as a (generalized) projection that satisfies nonsmooth function r. This argmin is called the *proximal operator*, written as

$$\text{prox}_{\eta_t r}(\tilde{\mathbf{w}}_{t+1}) \stackrel{\text{def}}{=} \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \tfrac{1}{2} \|\mathbf{w} - \tilde{\mathbf{w}}_{t+1}\|_2^2 + \eta_t r(\mathbf{w})$$

The final step is to understand when it is feasible to solve for this argmin. It is not easy for just any r. We either need a closed-form solution for this proximal operator or an efficient algorithm to solve for it. We will provide an example where a closed-form solution exists in Example 9 and an example where an iterative algorithm is needed in Appendix A.4.2.

Example 9: You have already seen one such r, for the interval constraint set $[-1, 1]^d$. Labeling the r for this constraint set as r_{box}, we get

$$\text{prox}_{\eta_t r_{\text{box}}}(\tilde{\mathbf{w}}_{t+1}) = \begin{bmatrix} \text{prox}_{\eta_t r_{\text{box}}}(\tilde{w}_{t+1,1}) \\ \text{prox}_{\eta_t r_{\text{box}}}(\tilde{w}_{t+1,2}) \\ \\ \text{prox}_{\eta_t r_{\text{box}}}(\tilde{w}_{t+1,d}) \end{bmatrix} \quad \text{where } \text{prox}_{\eta_t r_{\text{box}}}(v) \begin{cases} -1 & \text{if } v < -1 \\ v & \text{if } v \in [-1, 1] \\ 1 & \text{if } v > 1 \end{cases}$$

□

Others simple constraints that have closed-form proximal operators included non-negativity constraints, where $\text{prox}_{\eta_t r_{\text{pos}}}(v) = \max(v, 0)$, or constraining $\frac{1}{2}\|\mathbf{w}\|_2^2 \leq C$ for some constant C.

In the following two subsections, we will provide examples of two other closed-form proximal operators: one for ℓ_1 regularization and one for simplex constraints that ensure $\sum_{j=1}^m w_j = 1$ and $w_j \geq 0$ for mixture models.

6.2 Case Study: ℓ_1 Regularization for Feature Selection

Recall that ℓ_2 regularization corresponded to putting a prior on the weights. We mainly discussed using it with linear regression, but also discussed how it can be added to any GLM. This is similarly true for the ℓ_1 regularizer we discuss in this section. Again for simplicity, we primarily discuss these concepts for linear regression and leave it as an exercise to show how to use this regularization approach more generally for GLMs.

Let us revisit the linear regression objective, $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$. If we choose a Laplace distribution for the prior on the weights, we get an ℓ_1 penalized objective

$$c(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1$$

which is often called the Lasso. This objective can be obtained similarly to the ℓ_2 regularized objective, but instead using a Laplace distribution with parameter λ for the prior. As with the ℓ_2 regularizer for ridge regression, this regularizer penalizes large values in $\mathbf{w}$. However, it also produces more sparse solutions, where entries in $\mathbf{w}$ are zero. This preference can be seen in Figure 6.1, where the Laplace distribution is more concentrated around zero. In practice, however, this preference is even stronger than implied by the distribution, due to how the spherical least-squares loss and the ℓ_1 regularizer interact.

Forcing entries in $\mathbf{w}$ to zero has the effect of feature selection, because zeroing entries in $\mathbf{w}$ is equivalent to removing the corresponding feature. Consider the dot product each time a prediction is made,

$$\mathbf{x}^\top \mathbf{w} = \sum_{j=0}^{d} x_j w_j = \sum_{j:w_j \neq 0} x_j w_j.$$

This is equivalent to simply dropping entries in $\mathbf{x}$ and $\mathbf{w}$ where $w_j = 0$.

For the Lasso, we no longer have a closed-form solution. We do not have a closed form solution, because we cannot solve for $\mathbf{w}$ in closed-form that provides a stationary point. Instead, we use gradient descent to compute a solution to $\mathbf{w}$.

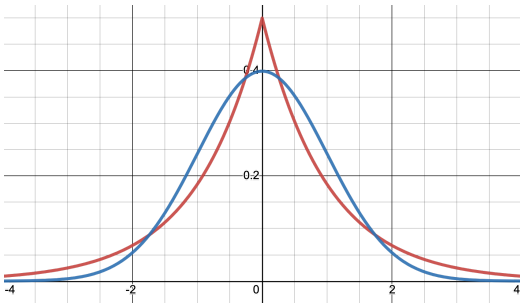


Figure 6.1: A comparison between Gaussian (blue) and Laplace (red) priors. The blue curve is $(2\pi)^{-1/2}\exp(-x^2/2)$*, which is the pdf for a* $\mathcal{N}(0,1)$*. The red curve is* $(1/2)\exp(-|x|)$*, which is the pdf for a Laplace with mean zero and* $b = 1$*. Both prefer values to be near zero, but the Laplace prior more strongly prefers the values to equal zero.*

The ℓ_1 regularizer, however, is non-differentiable at 0. We assume throughout these notes that our objectives are continuous. However, this need not mean that they are smooth: in some cases, these continuous objectives may have non-differentiable points. For example, the ℓ_1 regularizer is non-differentiable at 0, making $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1$ non-differentiable. One strategy is to use sub-gradient descent; loosely, this amounts to selecting a reasonable choice for the gradient at the non-differentiable point. Here, for example, we could take the partial derivative of ℓ_1 for w_j to be zero at zero, -1 for $w_j < 0$ and 1 for $w_j > 0$. Unfortunately, this descent is slow because there is a tendency to jump around zero. Unlike ℓ_2, the gradient does not gradually decrease near zero, slowly decreasing w_j, but rather jumps between two large values -1 and 1. With such large gradient, it is difficult to gradually decrease w_j to zero, even if that is the optimal solution.

One alternative for such non-smooth objectives is to use proximal methods. The idea is simple: use gradient descent for the smooth component of the optimization (the error term $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$), and then for values in $\mathbf{w}$ that are close to zero, set them to zero. This thresholding idea, though simple, is a theoretically sound approach for optimizing with the non-smooth ℓ_1. This thresholding operator is called the proximal operator, and can be seen as a projection operator. Each time $\mathbf{w}$ is updated with the gradient, it moves it away from a sparse solution; the proximal operator then projects $\mathbf{w}$ back onto the space of sparse solutions. The proximal operator for ℓ_1 is applied element-wise to $\mathbf{w}$, and so is defined on each $\mathbf{w}_i$ as, with stepsize η and regularization parameter λ,

$$\text{prox}_{\eta\lambda\ell_1}(w_j) = \begin{cases} w_j - \eta\lambda & \text{if } w_j > \eta\lambda \\ 0 & \text{if } |w_j| \leq \eta\lambda \\ w_j + \eta\lambda & \text{if } w_j < -\eta\lambda. \end{cases}$$

The proximal operator on the entire vector $\mathbf{w}$ is defined element-wise: $\text{prox}_{\eta\lambda\ell_1}(\mathbf{w}) = [\text{prox}_{\eta\lambda\ell_1}(w_1), \ldots, \text{prox}_{\eta\lambda\ell_1}(w_d)]$. Nicely, the theory states that the stepsize should be no larger than the inverse of the Lipschitz constant for the smooth part of the objective, where intuitively the Lipschitz constants reflects how quickly the function changes. In Algorithm 2, we provide a gradient descent algorithm for the incremental update with the ℓ_1 regularizer, introduced as an algorithm called ISTA [5]. More generally, proximal methods are used for other non-smooth objectives, though in these notes we only consider Lasso.

Algorithm 2: Batch gradient descent for ℓ_1 regularized linear regression $(\mathbf{X}, \mathbf{y}, \lambda)$

$\mathbf{w} \leftarrow \mathbf{0} \in \mathbb{R}^d$
err $\leftarrow \infty$
tolerance $\leftarrow 10e^{-4}$
// Precomputing these matrices, to avoid recomputing them in the loop
$XX \leftarrow \frac{1}{n}\mathbf{X}^\top\mathbf{X}$
$Xy \leftarrow \frac{1}{n}\mathbf{X}^\top\mathbf{y}$
// This stepsize is specific to the least-squares loss for linear regression
$\eta \leftarrow 1/(2\|XX\|_F)$
while $|c(\mathbf{w}) - \text{err}| >$ tolerance and have not reached max iterations **do**
    err $\leftarrow c(\mathbf{w})$
    // Proximal operator projects back into the space of sparse solutions given by ℓ_1
    $\mathbf{w} \leftarrow \text{prox}_{\eta\lambda\ell_1}(\mathbf{w} - \eta XX\mathbf{w} + \eta Xy)$
return $\mathbf{w}$

Feature selection is particularly pertinent when we have many features. You have already seen one setting where it is sensible: polynomial regression with a high-order polynomial. In Chapter 8 we will discuss other such fixed feature expansions for which feature selection can be useful.

6.3 Case Study: Simplex Constraints for Mixture Models

Another common constraint is the simplex constraint: $\mathcal{F} = \{\mathbf{w} \in [0,1]^m \mid \sum_{k=1}^m w_k = 1\}$. We have seen this constraint already for mixture models, where we need the components weights w_k to be a convex combinations to ensure we have a valid pdf: $p(\mathbf{x}|\mathbf{w}) = \sum_{k=1}^m w_k p_k(\mathbf{x})$. In other words, we have a simplex constraints on the weights: $\mathbf{w} \in \mathcal{F}$.

We can similarly consider how to do a proximal update under these constraints. Again, we get a descent direction $\tilde{\mathbf{w}}_{t+1}$ and then project (apply a proximal operator) to ensure these constraints are satisfied. This update is actually a bit complicated, so instead in the main notes we will show how we solve this constrained optimization for a different objective. Once you see this derivation, it is easier to understand the proximal update derivation, which we include in Appendix A.4.2. However, it is not key to specifically understand the proximal update, just to see how these tools are used. So the simpler derivation in this section serves our purpose and, when possible, it is useful to see ideas in their simplest form.

The simpler problem we consider is actually a useful one: it is the optimization we need to solve for mixture models in Chapter 11. In this section, we will look at how to solve for these weights, assuming given (fixed) components. Later, in Chapter 11, you will see how to also learn the parameters for the components. Our goal is to solve the following optimization problem

$$\mathbf{w}^* = \underset{\mathbf{w}\in[0,1]^m, \sum_{k=1}^m w_k = 1}{\text{argmin}} \; -\sum_{k=1}^m \text{probs}[k] \ln w_k$$

where $\text{probs}[k] \stackrel{\text{def}}{=} \sum_{i=1}^n p_k(\mathbf{x}_i)$ is a fixed array of likelihoods. We compute these likelihoods once, using the component pdfs, as in this section we are not learning parameters for those

components. We will later derive this objective when we talk about the MLE problem for mixture models, but here we will simply take it as a given.

To start we shall first form the Lagrangian function as

$$L(\mathbf{w}, a, \mathbf{b}) = -\sum_{k=1}^{m} p_t[k] \ln w_k + a\left(\sum_{k=1}^{m} w_k - 1\right) - \sum_{k=1}^{m} b_k w_k$$

where $a \in \mathbb{R}$ and $\mathbf{b} \geq \mathbf{0}$ are called the KKT multipliers. Our goal now is to solve the new objective that no longer has constraints on $\mathbf{w}$:

$$\min_{\mathbf{w}\in\mathbb{R}^m} \max_{a\in\mathbb{R}, \mathbf{b}\geq 0} L(\mathbf{w}, a, \mathbf{b})$$

We can find a closed form solution, by reasoning about the feasible solutions. We know that any optimal solution $\mathbf{w}$ must satisfy $\mathbf{w} \in \mathcal{F}$ (the simplex); otherwise, the loss $L(\mathbf{w}, a, \mathbf{b})$ can be made arbitrarily big by (the adversaries) a and $\mathbf{b}$. So, any optimal solution will have $\mathbf{w} \in \mathcal{F}$. Moreover, for such $\mathbf{w}$, we know that $a \in \mathbb{R}$ has no impact on the loss, since it multiples zero; therefore, it is a free variable and can be anything and still result in an optimal solution—namely result in the same value for $L(\mathbf{w}, a, \mathbf{b})$. Finally, we know that $\mathbf{b} \geq 0$ will be chosen such that $b_k w_k = 0$ for all k, since that is the choice that makes the sum involving $\mathbf{b}$ maximal. And, of course, we need to be at a stationary point for $\mathbf{w}$. In other words, to satisfy the KKT conditions and know we have an optimal solution, we need

$$\begin{aligned} 0 = \frac{\partial}{\partial w_k} L(\mathbf{w}, a, \mathbf{b}) &= -\frac{p_t[k]}{w_k} + a - b_k \qquad \forall k \in \mathcal{Y} \\ w_k b_k &= 0 \end{aligned}$$

with $\mathbf{b} \geq 0$ and $\mathbf{w} \in \mathcal{F}$. The stationarity conditions gives us $w_k = \frac{p_t[k]}{a - b_k}$. We know that $b_k = 0$ unless $w_k = 0$. If $p_t[k] > 0$, we know $w_k > 0$, giving $w_k = \frac{p_t[k]}{a}$. If $p_t[k] = 0$, then $w_k = 0$ and similarly we can write $w_k = \frac{p_t[k]}{a}$. Therefore, to ensure we are at an optimal solution—and satisfy the KKT conditions—we know we need to set a such that $\sum_{k=1}^{m} w_k = 1$. We can do so by setting $a = n$, giving

$$\sum_{k=1}^{m} w_k = \sum_{k=1}^{m} \frac{p_t[k]}{n} = \frac{1}{n} \sum_{k=1}^{m} p_t[k] = \frac{1}{n} n = 1$$

because by definition $\sum_{k=1}^{m} p_t[k] = n$. Therefore,

$$w_k^{(t+1)} = \frac{1}{n} p_t[k] \quad \text{where} \quad p_t[k] \stackrel{\text{def}}{=} \sum_{i=1}^{n} p_t[i, k] \tag{6.1}$$

It may feel weird that we had to do so much reasoning to find the solution. Usually, we just do gradient descent and declare success. We actually could have used iterative methods like gradient descent to solve this Lagrangian optimization over $\mathbf{w}$, a and $b \geq 0$. But, why use an iterative method when we can get a closed-form solution. Here, we were actually able to reason our way to a closed-form solution, which is as good a way as any. In the end, what matters is that we got our closed-form solution.

Chapter 7

Evaluating Generalization Performance

Before deploying a learned model f, we want to obtain an estimate of its generalization performance: its expected error. Ideally, we would train f on all the available data, to facilitate identifying as good a model as possible. However, performance on the training set is highly biased: it is likely to be much lower than the true generalization error, since the model was trained to minimize it. In many cases, with complex models like neural networks, it is common to get zero error on the training set. We had previously discussed using a hold-out test set, to obtain an unbiased estimate of the generalization error. This remains a relatively common approach, but as we discuss in this chapter, it might be worth introducing some bias to obtain a more accurate estimate of the generalization error.

7.1 Defining Generalization Error

Machine learning is all about generalization. We learn a model on a sample (subset) of possible outcomes, with the goal for it to be accurate across all possible outcomes. Accuracy here is defined based on the chosen cost function $\text{cost} : \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$. We want to find a function f so that the generalization error GE (expected cost) is minimal

$$\text{GE}(f) = \mathbb{E}[\text{cost}(f(X), Y)]$$

For example, for regression with a squared error, $\text{cost}(\hat{y}, y) = (\hat{y} - y)^2$ and $\text{GE}(f) = \mathbb{E}[(f(X) - Y)^2]$. We may obtain such an f by minimizing the squared errors on a training set $\frac{1}{n}\sum_{i=1}^{n}(f(\mathbf{x}_i) - y_i)^2$ or by minimizing a regularized objective like one with an ℓ_2 or ℓ_1 regularizer.

As another example, for classification, we may want to minimize the 0-1 cost, namely

$$\text{GE}(f) = \mathbb{E}[\mathbb{I}\left(f(X) \neq Y\right)].$$

We may do so by learning $p(y|\mathbf{x})$ with logistic regression and specifying $f(\mathbf{x}) = 1$ if $p(y|\mathbf{x}) > 0.5$ and $f(\mathbf{x}) = 0$ otherwise. Or, in other words, if $p(y|\mathbf{x}) = \sigma(\boldsymbol{\phi}(\mathbf{x})^\top \mathbf{w})$ for some features $\boldsymbol{\phi}$ and weightings $\mathbf{w}$, then $f(\mathbf{x}) = \text{sign}(\boldsymbol{\phi}(\mathbf{x})^\top \mathbf{w})$. For this cost, we use a surrogate—the cross-entropy loss—rather than directly optimizing the 0-1 cost. We motivated this when considering ideal predictors: the best predictor for the 0-1 cost is to use the most likely class under the true model $p_{\text{true}}(y|\mathbf{x})$. A reasonable choice is to approximate this ideal predictor, by approximating $p_{\text{true}}(y|\mathbf{x})$.

Exercise 23: Consider instead trying to directly optimize the 0-1 cost on a training set: $\frac{1}{n}\sum_{i=1}^{n}\mathbb{I}\left(f(\mathbf{x}_i) \neq y_i\right)$. What is the issue? What does this loss look like and what are the derivatives? □

7.2 Estimating Generalization Error using Cross Validation

Assume you are given a dataset $\mathcal{D}$ and you learn a predictor $f_{\mathcal{D}} : \mathcal{X} \to \mathcal{Y}$ using your favourite regression algorithm. You'd like to know what the true generalization error is for your model, in terms of squared errors

$$\mathrm{GE}(f_{\mathcal{D}}) = \mathbb{E}[(f_{\mathcal{D}}(X) - Y)^2 | \mathcal{D}]$$

where we explicitly write that the data is given to emphasize it is not random in this definition. Naturally, if we had another batch of data with m samples—let's call it test data $\mathcal{D}_{\text{test}}$—then we could use a sample average to get an unbiased estimate of $\mathrm{GE}(f_{\mathcal{D}})$

$$\mathrm{GE}(f_{\mathcal{D}}) \approx \frac{1}{m} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}_{\text{test}}} (f_{\mathcal{D}}(\mathbf{x}_i) - y_i)^2$$

The larger $\mathcal{D}_{\text{test}}$ is, the closer the approximation. In fact, using concentration inequalities, we know this should get closer at a rate of $m^{-1/2}$ for $m = |\mathcal{D}_{\text{test}}|$.

The dilemma is that you would like to use this test data to learn a better model. Even in this age of huge datasets, we still typically want to learn on as much (quality) data as possible. Our goal is to get a good estimate of performance, without having to put aside too much test data or even none at all. In this section, we talk about how *cross validation* provides one approach to get a biased but nonetheless reasonable estimate of the performance of the model trained on the whole dataset. We will talk about two variants of cross-validation, which only differ in how they generate subsamples: *k-fold cross validation* and *repeated random subsampling*.

To be precise, we assume that we train $f_{\mathcal{D}}$ on the whole dataset of n samples. We do not split the dataset into train and test. We will then use the same dataset to estimate $\mathrm{GE}(f_{\mathcal{D}})$, by actually measuring the errors for several different f learned on different subsets of the $\mathcal{D}$. But how can we do this amazing thing, you ask? Let me tell you.

The general idea is the same for k-fold cross-validation and repeated random subsampling. The only difference is in how we obtain these subsets. For now, let us assume we use repeated random subsampling; we will describe k-fold cross validation later. Either of these two approaches produces k different subsets of the data, $(\mathcal{D}_{\text{tr}}^{(1)}, \mathcal{D}_{\text{te}}^{(1)}), \ldots, (\mathcal{D}_{\text{tr}}^{(k)}, \mathcal{D}_{\text{te}}^{(k)})$, where $\mathcal{D}_{\text{tr}}^{(j)} \cup \mathcal{D}_{\text{te}}^{(j)} = \mathcal{D}$ and $\mathcal{D}_{\text{tr}}^{(j)} \cap \mathcal{D}_{\text{te}}^{(j)} = \varnothing$. In other words, each train-test split has disjoint training and test data and together consists of all the data. Repeated random subsampling generates a train-test split by randomly sampling (without replacement) a subset of the data for training and then using the remaining samples for test.

The estimate of generalization error is obtained as follows. Let $(f^{(1)}, \mathsf{err}^{(1)}), \ldots, (f^{(k)}, \mathsf{err}^{(k)})$ be the corresponding learned functions and test errors, where $f^{(j)}$ is trained on $\mathcal{D}_{\text{tr}}^{(j)}$ and $\mathsf{err}^{(j)}$ is the error of $f^{(j)}$ on $\mathcal{D}_{\text{te}}^{(j)}$. This procedure mimics training $f_{\mathcal{D}}$ and then testing it, in deployment, on new unseen data. The primary difference is that each training set is a bit smaller than the actual training set used by $f_{\mathcal{D}}$, since each $\mathcal{D}_{\text{tr}}^{(j)}$ uses only a subset of $\mathcal{D}$. The corresponding errors are reasonably reflective of the error we might expect to see for $f_{\mathcal{D}}$, at least on average. The estimator we use is

$$\mathrm{GE}(f_{\mathcal{D}}) \approx \bar{G} \stackrel{\text{def}}{=} \frac{1}{k} \sum_{j=1}^{k} \mathsf{err}^{(j)}$$

This entire procedure is depicted in Figure 7.1.

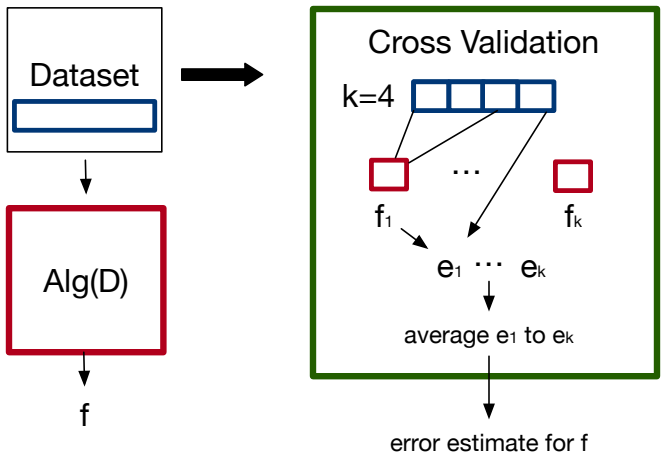


Figure 7.1: A depiction of cross validation. The dataset is colored in blue, with each of the 1 to k partitions shown as a smaller cube to indicate they are a subset. The algorithm is labeled in red, where it is called on different subsets of the data ($k-1$ of the partitions). We assume here that the algorithm just needs to input the dataset, and has its own mechanism to pick any hyperparameters. The errors for each of the learned $f_1, \ldots, f_k$ are computed on the held-out partition, resulting in errors $e_1, \ldots, e_k$. The average of these provides our CV estimate of the error for the function f, which is learned on the entire dataset.

7.3 Bias and Variance of the Cross Validation Estimator

How reliable is our cross-validation (CV) estimator? And how do we select parameters like the number of folds k for this estimator? In this section, we reason about the bias and variance of the CV estimator, to better understand how accurately it estimates the generalization error.

The CV estimator has two sources of stochasticity: the randomness when sampling the dataset $\mathcal{D}$ and then randomness from creating the partitioning. First we sample a random dataset $\mathcal{D}$ from the true joint model p. Then we randomly partition the data, and get samples of error obtain a partitioning j, to get sample $\mathsf{err}^{(1)}, \mathsf{err}^{(2)}, \ldots, \mathsf{err}^{(k)}$. We can reason about the bias and variance of the CV estimator over these two sources of randomness.

The CV estimator is biased because $\mathsf{err}^{(j)}$ is likely a pessimistically biased estimate of the true error of $f_{\mathcal{D}}$. This bias arises because $\mathsf{err}^{(j)}$ is the error for a function learned using the same algorithm on a smaller dataset. More data typically means we can learn a more accurate function, and so we expect the error, in expectation, to be higher for each $f^{(j)}$ than $f_{\mathcal{D}}$.

The amount of bias depends on (1) the size of the training sets in the partitioning and (2) the size of the dataset. If most of the data is used in the training set, then $f^{(j)}$ is similar to $f_{\mathcal{D}}$ and so there is little bias. In the most extreme case, all but one sample can be used for $\mathcal{D}^{(j)}_{\text{tr}}$ and $\mathcal{D}^{(j)}_{\text{te}}$ consists of only one point (called *leave-one out*). On the other hand, if most of the data is used in the test set, then the bias is bigger. Most of this is not an issue as the size of the dataset n gets big. The bias disappears as n becomes very big because the functions should become very similar to each other even if data is evenly partitioned

between training and test.

We can also reason about the variance of this estimator.

$$\text{Var}\left[\bar{G}\right] = \frac{1}{k^2}\left(\sum_{j=1}^{k}\text{Var}\left[\mathsf{err}^{(j)}\right] + \sum_{i,j}\text{Cov}[\mathsf{err}^{(i)}, \mathsf{err}^{(j)}]\right)$$

The variance decreases if (1) k is larger and (2) $\text{Var}\left[\mathsf{err}^{(j)}\right]$ is smaller, which occurs when n is larger. Notice that (2), however, is also dependent on how many samples are used for training and for test. The relationship is nuanced, as the variance of the errors might be higher for a small test set, but the variance of the learned function might be higher with a smaller training set. Further, if k becomes larger, typically there is more correlation between each $(f^{(i)}, \mathsf{err}^{(i)})$ and $(f^{(j)}, \mathsf{err}^{(j)})$, making the covariance term larger. Overall, due to these nuanced relationships,[1] there is not a clear answer for how to choose k and the partitioning. There are, however, some rules of thumb, which we discuss after Exercise 24.

Exercise 24: We previously discussed that once a hold-out test set has been used for evaluation, we cannot use it again because it will not provide an unbiased estimate of the expected error. For example, after getting performance of your models on that test set, one could go back and adjust hyperparameters such as the regularization parameters. However, once you have done this, the test-set has influenced the learned models and is likely to produce an optimistic estimate of performance on new data. Is this also true for cross validation? Namely, if you realized you should have tested a model with different hyperparameters, and reran the cross validation procedure for the new model, would your estimate using cross validation suffer from similar bias as in the hold-out test set case? □

Now let us revisit the strategies to generate these sub-datasets. In k-fold cross-validation the data is partitioned into k disjoint sets, called folds. The training set is composed of $k-1$ of the folds, and test set is the remaining fold. We use all possible combinations—namely each fold is used once as a test set—resulting in k train-test splits. This partitioning approach has the advantage that the resulting k performance estimates are mostly independent, with some dependency introduced due to dependencies between the training sets. As mentioned above, there is also the bias from the fact that we do not run the model on the entire training set, but rather get an estimate of the error for the algorithm trained on $n-(n/k)$. The disadvantage of this approach is that the number k both dictates how many train-test splits we consider as well as the size of the training and test sets.

Repeated random subsampling (RRS) does not suffer from that same issue, but has its own disadvantages. In RRS, because we create splits using random sampling rather than a disjoint partitioning, we can decouple k and the size of the training and test sets. For example, we might want to have at least $k = 10$ different train-test splits, but we might want to use more than $n-(n/k)$ for train; RRS allows this. You could take a dataset of size 1000, set $k = 10$ and pick the training set size to be 950 and 50 for test. In k-fold CV, once you pick $k = 10$, the size of the training set is set to 900 and test is set to 100.

There are common rules of thumb to pick these sizes. A common choice for k-fold CV is to use $k = 10$. For RRS, it is generally reasonable to pick k a bit higher, though a limiting factor is always computation, since you need to train the model k times to get the error estimate. For slow deployment settings—where the predictor in deployment is changed

[1] See Appendix A.2 for a just alittle bit more discussion on some of these nuances

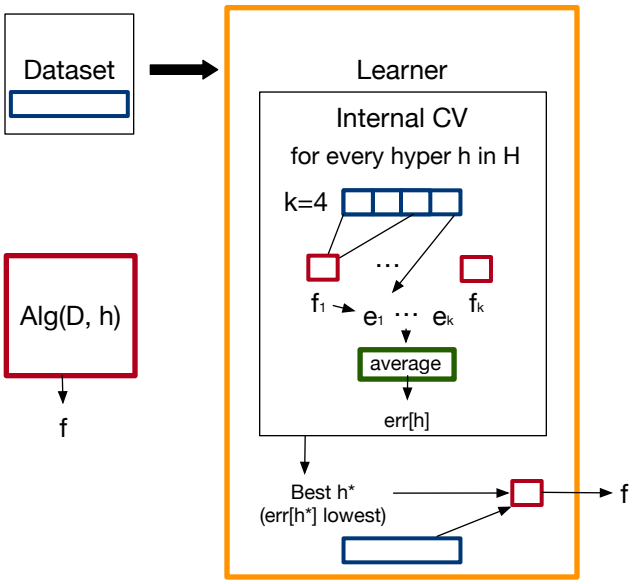


Figure 7.2: A depiction of internal cross validation. The dataset is colored in blue, with each of the 1 to k partitions shown as a smaller cube to indicate they are a subset. The algorithm is labeled in red, where it is called on different subsets of the data ($k-1$ of the partitions) for a given hyperparameter h. The errors for each of the learned $f_1, \ldots, f_k$ are computed on the held-out partition, resulting in errors $e_1, \ldots, e_k$. The average of these provides our CV estimate of the error for the function that would be learned using that hyperparameter h on the entire dataset. Internal CV then picks the h^* that has the lowest error. The output of internal CV is that h^*. Then the learner takes the h^* and the entire dataset and output the final learned function f.

infrequently or we need to be very careful about deploying any predictor—the cost of this offline evaluation is not too limiting of a factor. For faster turnaround times—say repeated testing and deployment—a large k may become prohibitive.

To better understand the properties of these two approaches, see the thorough and accessible explanation in [12, Chapter 5].

7.4 Using Cross Validation to Select Hyperparameters

You have seen how to gauge the generalization performance of a model, using validation sets. We can leverage the exact same idea to select hyperparameters. Why? Because our criteria is exactly the same: we want to select hyperparameters that result in the best generalization performance. The idea is simple: if there are m possible hyperparameter settings, then evaluate m models each corresponding to a hyperparameter setting, and select the hyperparameter setting that has the best evaluation performance.

This procedure is called *internal cross-validation*, because it is internal to the algorithm. It is part of the algorithm, because it is the way that the algorithm sets its hyperparameters. For example, we can consider ridge regression with internal CV to set the regularization parameter as one complete algorithm, that returns a model when run on a given dataset.

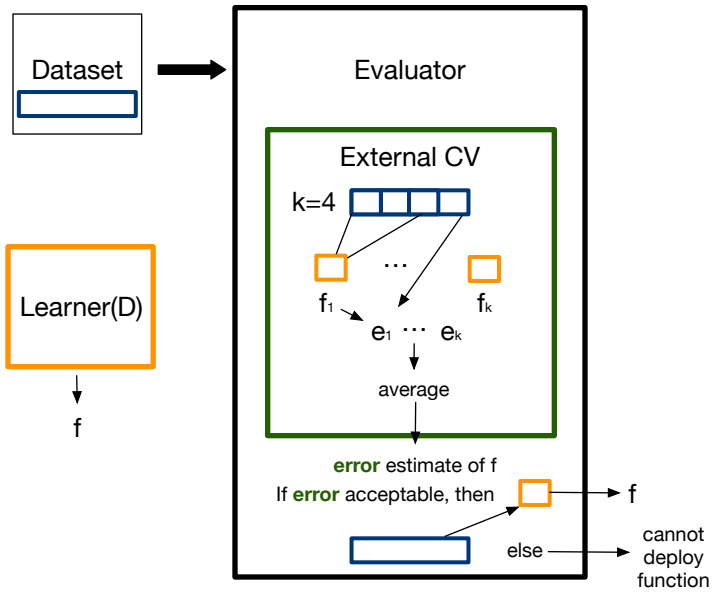


Figure 7.3: A depiction of external cross validation. The dataset is colored in blue, with each of the 1 to k partitions shown as a smaller cube to indicate they are a subset. The learner is labeled in orange, just as in Figure 7.2, where it is called on different subsets of the data ($k-1$ of the partitions) for a given hyperparameter h. This is a key difference from internal CV. The errors for each of the learned $f_1, \ldots, f_k$ are computed on the held-out partition, resulting in errors $e_1, \ldots, e_k$. The average of these provides our CV estimate of the error for the function that is learned by our Learner on the entire dataset. (The learner itself may use internal CV). Then the Evaluator decides, based on the error estimate of f given by external CV, whether it is happy to deploy f.

This is visualized in Figure 7.2.

We can then take this model and evaluate it using the techniques described earlier in this chapter. If we use cross-validation to evaluate this model, then we call this *external cross-validation* to disambiguate. To separate the two roles, we can think of this settings as having an *evaluator* and a *learner*. The learner's job is to return a model when it is given a dataset. The evaluator's job is to obtain a model for deployment, and so needs to provide an estimate of the performance of that model.

The full procedure is as follows, depicted also in Figure 7.3.

1. The evaluator is given a dataset $\mathcal{D}$.
2. The evaluator obtains a model $\boldsymbol{\theta}_{\mathcal{D}}$ by running the learner on the dataset $\mathcal{D}$.
3. Before deploying that model $\boldsymbol{\theta}_{\mathcal{D}}$, the evaluator obtains an estimate of the performance of $\boldsymbol{\theta}_{\mathcal{D}}$ using (external) cross-validation. For example, the evaluator might use k-fold CV with $k = 5$, calling the learner fives times on each subset of data.
4. Each time the learner is called on a dataset $\mathcal{D}'$—labeled differently than $\mathcal{D}$ to indicate that this dataset is a subset from external CV—it needs to return a model $\boldsymbol{\theta}_{\mathcal{D}'}$. To do so, it needs to decide on its own hyperparameters, like the regularization parameter

Algorithm 3: Nested cross-validation on a dataset $\mathcal{D}$

Partition the dataset $\mathcal{D}$ into k_{external} folds
Initialize err-f $= 0$
for $i = 1$ to k_{external} **do**
  Set $\mathcal{D}_{\text{te}}^{(i)}$ to the data in fold i
  Set $\mathcal{D}_{\text{tr}}^{(i)} = \mathcal{D} - \mathcal{D}_{\text{te}}^{(i)}$
  $f_i \leftarrow$ **Learner**$(\mathcal{D}_{\text{tr}}^{(i)})$
  err-f $=$ err-f$+$ error of f_i on $\mathcal{D}_{\text{te}}^{(i)}$
err-f $=$ err-f$/k_{\text{external}}$
$f_i \leftarrow$ **Learner**$(\mathcal{D}_{\text{tr}}^{(i)})$
return f and err-f

Algorithm 4: Learner using cross-validation on a dataset $\mathcal{D}'$

Partition the dataset $\mathcal{D}'$ into k_{internal} folds
for h in the set of hyperparameters H **do**
  Initialize err$[h] = 0$
  for $j = 1$ to k_{internal} **do**
    Set $\mathcal{D}'^{(j)}_{\text{te}}$ to the data in fold j for dataset $\mathcal{D}_{\text{tr}}^{(i)}$
    Set $\mathcal{D}'^{(j)}_{\text{tr}} = \mathcal{D}' - \mathcal{D}'^{(j)}_{\text{te}}$
    Train $f = \text{Alg}(\mathcal{D}'^{(j)}_{\text{tr}}, h)$
    err$[h] =$ err$[h]+$ error for f on $\mathcal{D}'^{(j)}_{\text{te}}$
  err$[h] =$ err$[h]/k_{\text{internal}}$
Pick $h^* = \text{argmin}_{h \in H}\, \text{err}[h]$
// Learner done picking its hyperparameter, can now return the learned function
Train $f = \text{Alg}(\mathcal{D}, h^*)$
return f

λ. It uses internal CV on $\mathcal{D}'$. For example, it might use k-fold CV on $\mathcal{D}'$ to evaluate each possible hyperparameter.

In Algorithm 3, in pseudocode, we more explicitly write what is called nested cross-validation, where the evaluator use k-fold cross-validation and the learner also uses k-fold cross-validation. The evaluator is said to use external CV, because it is in the outer loop, and the learner is said to use internal CV because it is in the inner loop. We have separated out the **Learner** pseudocode in Algorithm 4 because it is used both inside the for loop and again at the end of the algorithm to learn the final function.

This pseudocode helps us reason about the computational complexity. If we copied in the Learner function into our for loop in Algorithm 3, then we would see three for loops: external loop over folds, a loop over hyperparameters and then an internal loop over folds. In total, this means we can the algorithm $k_{\text{external}} k_{\text{internal}} |H|$ times, where $|H|$ is the number of hyperparameters. This can be very expensive. For example, if we use 10 folds for external CV, with 8 hyperparameter choices and 10 internal folds, we call the algorithm 800 times!

The criteria for making choices may differ between internal and external CV. For external CV, the evaluator wants to have a high confidence estimate of performance, and

will likely use confidence intervals obtained from the CV. The evaluator might prefer to use Monte Carlo CV, because the repeated resampling allows for more estimates of performance without training on too small of datasets. For internal CV, on the other hand, the agent does not need high confidence estimates. Rather, it simply needs to obtain a reasonable hyperparameter. If computation is an issue—we may not want our algorithm to be too slow—then we might pick a smaller k. The learner may want to ensure that it systematically covers the data points and might be worried that random resampling with a small k might not do so. It might therefore opt for k-fold CV for internal CV.

Exercise 25: Imagine you use $k = 2$ CV for internal CV, when selecting λ for ridge regression. What issue might there be with selecting such a small k? □

Exercise 26: Notice that at the end of nested CV we call the Learner on all of the data, $\mathcal{D}$. This step also produces the best hyperparameters h^* on $\mathcal{D}$, though the Learner does not return this h^*, it only returns the corresponding function. But it seems like we can avoid nested CV altogether by simply using this h^*. In other words, we use CV on $\mathcal{D}$ to find hyperparameters h^*. Then we fix these hyperparameters, and do CV on $\text{Alg}(\cdot, h^*)$. This procedure only costs $k_{\text{internal}}|H| + k_{\text{external}}$, rather than $k_{\text{external}}k_{\text{internal}}|H|$. In our above example, this reduces the number of times we call the algorithm to $10 \times 8 + 10 = 90$, as opposed to 800. Though this is in fact a relatively common approach, because of this significant computational savings, it is likely more biased that nested CV. Explain why. □

Nested cross-validation can be expensive, so do we have to do it? It seems like we should be able to do a two-staged approach instead, where we first select hyperparameters using CV, and then we do CV again for the model with these hyperparameters. But, this is likely to produce a biased result, one that makes our model appear better than it is. It is a little bit like picking hyperparameters using a test set, selecting the best ones, and then using the test set again to evaluate the model with those hyperparameters. Because we picked the hyperparameters that are best for the test set, we effectively used the test set for training and we have slightly fit to it. In CV, we at least use different randomizations of the data when we do CV in the first phase to pick hyperparameters and in the second phase when we evaluate the model with those hyperparameters. This two stage approach even with the randomization in CV is likely more biased optimistically, because we used CV to select hyperparameters and so the second stage of CV will also likely report higher performance for those same hyperparameters. There is some work showing that this seemingly benign modification, of moving from nested cross-validation to this two stage approach, can result in significant bias [9]. If you choose to do it, use it with caution!

Remark: Note that CV is not the only way to select hyperparameters; it is simply a commonly used, relatively generic approach. For certain hyperparameters, there are specialized strategies or rules of thumb. In other cases, the goal of algorithm development is to provide approaches that adapt the hyperparameter, where the newly introduced hyperparameters for the algorithm are less sensitive and easier to set. One such example is the stepsize selection rules you have seen.

Part II

Data Representations

At first, it might seem that the applicability of linear regression and logistic regression to real-life problems is greatly limited. After all, it is not clear whether it is realistic (most of the time) to assume that the target variable is a linear combination of features. Fortunately, the applicability of linear regression—and generalized linear models— is broader than it seems at first glance. The main idea is to apply a non-linear transformation to the data matrix $\mathbf{x}$ prior to the fitting step, which then enables a non-linear fit. Obtaining such a useful *data representation* is a central problem in machine learning.

We have already seen one instance of such a transformation, that allowed for nonlinear functions: polynomials, in polynomial linear regression. More generally, there are many other fixed representations for regression. Two common ones are radial basis function (RBF) networks and prototype representations. Even more common, however, is to learn the data representation, rather than simply specifying a fixed one. We first discuss fixed data representations and what they provide. Then we discuss learning data representations, primarily with factor analysis and neural networks.

Chapter 8

Fixed Representations

In this chapter we discuss two fixed representations, and why we might want to use them. First, however, we start by discussing why projecting into a higher dimension can allow us to learn more complex functions for regression and more easily separate classes for classification.

8.1 The Utility of Projecting to Higher Dimensions

Let us motivate the utility of projecting to higher dimensions using classification. In particular, we will see that it will allow us to get linear separability. Recall that linear separability means that we can perfectly separate the two classes using a linear hyperplane. An example of this is given in Figure 8.1.

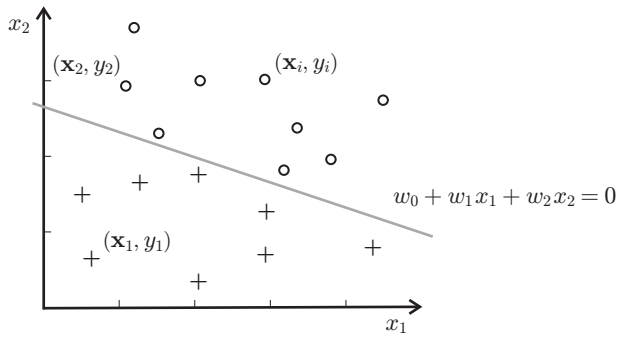


Figure 8.1: An example of a linearly separable dataset, with positive points labeled at + and negative as circles. The learned linear classifier (using logistic regression) can perfectly classify the points in this dataset. The decision boundary is given by the equation of the line $\mathbf{xw} + w_0 = 0$. *These are all the points* $\mathbf{x} = (x_1, x_2)$ *that are orthogonal to* $\mathbf{w} = (w_1, w_2)$. *Here picture a unit vector* $\mathbf{w}$ *pointing orthogonally to this line, towards the positive points, with* w_0 *giving the offset of this line away from zero. Any point* $\mathbf{x}$ *below the line is no longer orthogonal, and in fact has an angle to* $\mathbf{w}$ *that is less than 90 degrees (starting to point more in the direction of* $\mathbf{w}$*). This tells us this* $\mathbf{x}$ *is labelled as positive. The opposite is true for* $\mathbf{x}$ *above the line, which have more than a 90 degree angle to* $\mathbf{w}$*; some are even pointing in the opposite direction to* $\mathbf{w}$ *(i.e.,* $\mathbf{x} = -\mathbf{w}$ *would be classified as negative). This relationship is easy to see with the following cosine formula:* $\mathbf{xw} = \|\mathbf{x}\|\|\mathbf{w}\| \cos\theta$ *for the angle* θ *between the two vectors. The term* $\cos\theta$ *is positive for* $\theta \in [0, 90)$ *degrees and negative for* $\theta \in (90, 180]$.

Usually, however, we will not be able to linearly separate the data. Instead, we might have something like the nonlinear relationship in Figure 8.2. But, we can see that it should

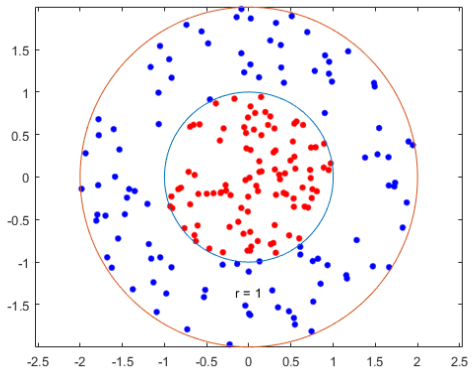


Figure 8.2: An example of a dataset that is not linearly separable. The function that separates these points is $f(\mathbf{x}) = x_1^2 + x_2^2 - 1$. For example, $f([0,0]) = -1 < 0$, which is correct as the red points in the inner ring are negative). And $f([2,-1]) = 4 + 1 - 1 = 4 > 0$, which lies within the positive blue points in the outer ring.

be easy to classify these points. In fact, we can make this dataset linearly separable by changing the features. We started with a two-dimensional input x_1, x_2. We want a function $f(\mathbf{x})$ where when $f(\mathbf{x}) < 0$ we classify the point as negative and when $f(\mathbf{x}) \geq 0$, we classify the point as positive. The function that works for this example is $f(\mathbf{x}) = x_1^2 + x_2^2 - 1$.

We could have easily learned a linear function that produced this f if we had first created a new set of three-dimensional features $\boldsymbol{\phi}(\mathbf{x}) = [x_1^2, x_2^2, 1]$ and learned a linear function in this new space. In other words, in a space one-dimension higher, this dataset is again linearly separable. By projecting to a higher-dimension, we can linearly separate the data.

More generally, moving up one dimension is not enough. But intuitively once you give yourself enough dimensions you get a lot more flexibility to place a line that properly separates points. There is even a formal statement for this, called Cover's Theorem, that says that a dataset that is not linearly separable is highly likely to be separable by projecting to a higher-dimensional space with a nonlinear transformation. One such example is given in Figure 8.3 showing how binning can be one such way to obtain this separation.

A key question is how to pick this projection to higher dimensions: how to pick this data representation. That is the focus of both fixed representations, discussed in this chapter, as well as learned representations discussed in the next chapter. We typically have at least two criteria for the choice. One is about sufficient *complexity*: did we expand the function space enough to get close to representing the true function (reduce bias). And another is *compactness* (a minimal representation), to make learning more sample efficient and computationally efficient. We may also care about properties of the features themselves and how they interact with the learning algorithm. For example, orthogonal features might make learning more effective when using stochastic gradient descent, whereas this might not matter as much for second-order gradient descent.

8.2 Radial Basis Function Networks

Radial basis functions (RBF) provide nonlinearity, just like polynomial transformations. We first explain what they are and then explain the types of functions they allow us to

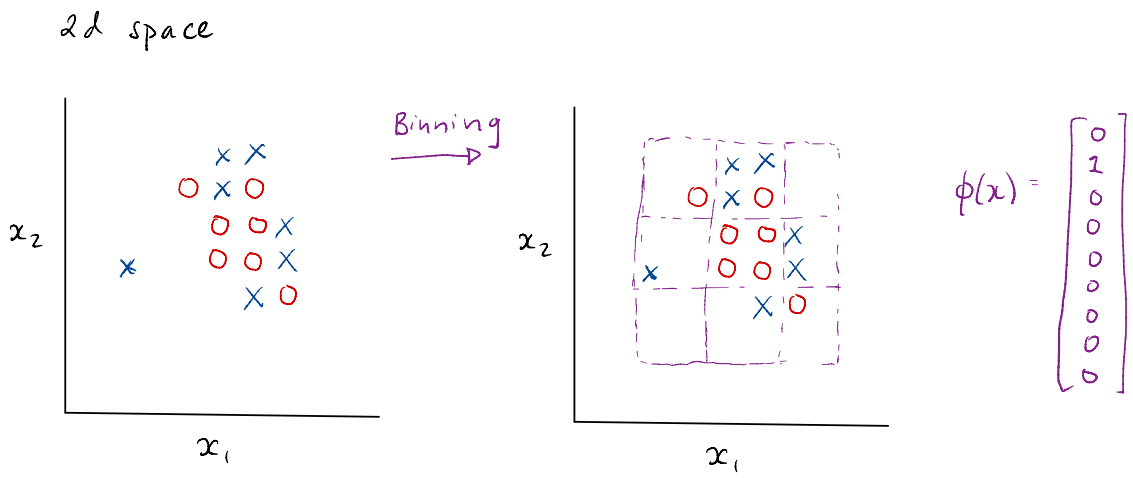


Figure 8.3: This 2d space is project up into a 7d space, by binning. The blue x's and red o's are not separable in the 2d space, and actually even in 7d they are not yet separable either (though there may be a smarter transformation to 7d that would give separability). But in the 7d space the accuracy of the classifier is now much better. Every point in a bin has to be classified the same, so the top middle bin will be classified as blue x's even though we have one red o in that bin. But every other bin has perfect accuracy. Trivially, we could have added another bin—or even one bin for every point—and got separability.

represent.

As usual, assume we are given data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$. We start by picking p points in $\mathcal{X}$ to serve as the *centers*. We denote those centers as $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_p \in \mathcal{X}$. These can be selected in a variety of ways, with some of the most common including

1. to uniformly cover $\mathcal{X}$
2. selected as a subset from $\mathcal{D}$ or
3. computed using some clustering technique on the data, such as k-means clustering.

The resulting basis functions—giving a fixed representation—consist of p new features $\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_p(\mathbf{x})$ with $\phi_j(\mathbf{x}) \in \mathbb{R}$ where

$$\phi_j(\mathbf{x}) \stackrel{\text{def}}{=} \exp\left(-\tfrac{1}{2\sigma^2}\|\mathbf{x} - \mathbf{c}_j\|_2^2\right),$$

where σ gives the width of the Gaussians. The resulting features are between 0 and 1. A feature $\phi_j(\mathbf{x})$ is close to one if $\mathbf{x}$ is similar to $\mathbf{c}_j$, and close to zero if its dissimilar. The size of σ determines how many features are non-negligibly large. If σ is very small, then most features are near zero; if σ is very large, then an input $\mathbf{x}$ has a non-negligible value for many centers. This collection of features is often called an RBF network, because we can view it as a network where the first layer transforms the input and then weights are learned on these new features, as in Figure 8.4.

As with polynomial features, for this new fixed set of features, we can simply apply any generalized linear model to obtain a nonlinear predictor. The feature transformation

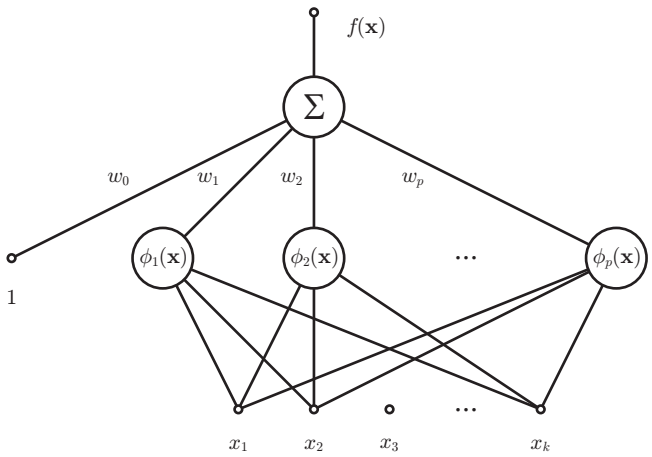


Figure 8.4: Radial basis function network.

provides nonlinearity, and we can exploit the simplicity of the linear methods to find the weights on the given dataset.

$$\mathbf{\Phi} = \begin{bmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_p(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_p(\mathbf{x}_2) \\ \vdots & & \ddots & \\ \phi_0(\mathbf{x}_n) & & & \phi_p(\mathbf{x}_n) \end{bmatrix}$$

is now used as a new data matrix. For a given input $\mathbf{x}$, the prediction of the target y will be calculated as

$$f(\mathbf{x}) = w_0 + \sum_{j=1}^{p} w_j \phi_j(\mathbf{x}) = \sum_{j=0}^{p} w_j \phi_j(\mathbf{x}) = \boldsymbol{\phi}(\mathbf{x})\mathbf{w}$$

where $\phi_0(\mathbf{x}) = 1$ and $\mathbf{w}$ is found using linear regression. More generally, for any GLM with associated transfer g, we use $g(\boldsymbol{\phi}(\mathbf{x})\mathbf{w})$ for the prediction.

Now we can ask why we would use this transformation. First, it provides *complexity* or *capacity*. It can be proved that with a sufficiently large number of radial basis functions we can accurately approximate any function [20]. In other words, it allows us to increase the hypothesis space that we consider for learning, such that we can include the true function underlying the data. In this way, we can learn a broader set of functions, while still exploiting the simplicity of our linear algorithms.

The additional nuance for using RBFs is appropriately selecting hyperparameters, like the variance σ and the centers. We had this problem too with polynomial features, where we needed to decide on the degree of the polynomial. A typical solution is to select σ using cross-validation, which we discussed in Section 7.4, though there are a variety of rules of thumb based on the number of centers and distances between each center. For example, one choice is: for each center, find the closest center and select the width σ to be twice that distance.

The update for linear regression using these new features is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \left(\boldsymbol{\phi}_i \mathbf{w}_t - y_i\right) \boldsymbol{\phi}_i$$

where $\boldsymbol{\phi}_i \stackrel{\text{def}}{=} \boldsymbol{\phi}(\mathbf{x}_i) = [\phi_1(\mathbf{x}_i), \phi_2(\mathbf{x}_i), \ldots, \phi_p(\mathbf{x}_i)]$. More generally, for any GLM with transfer g, the update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \left(g(\boldsymbol{\phi}_i \mathbf{w}_t) - y_i\right) \boldsymbol{\phi}_i.$$

These models are no longer linear in the original inputs, but rather only in the new features. The implicit assumption now is that we learn a model such that

1. $\mathbb{E}[Y|\mathbf{x}] = g(\boldsymbol{\phi}(\mathbf{x})\boldsymbol{\omega})$
2. $p(y|\mathbf{x})$ is an Exponential Family distribution, with parameter $\theta(\mathbf{x}) = \boldsymbol{\phi}(\mathbf{x})\boldsymbol{\omega}$.

8.3 Prototype Representations

RBF networks and prototype representations are highly related. The main distinction is that prototype representations use any similarity measure $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ to produce the feature

$$\phi_j(\mathbf{x}) \stackrel{\text{def}}{=} k(\mathbf{x}, \mathbf{c}_j).$$

Radial basis functions are one example of such a similarity k. In addition, for prototype representations the points $\mathbf{c}_j$ are typically subselected from the training dataset; these $\mathbf{c}_j$ are prototypical instances, thus the name prototype representations. More formally, we pick p points $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_p$ *from our dataset*, without replacement. If n is sufficiently small, it is common to simply use $p = n$; otherwise, these prototypes are chosen to be a diverse set. For RBF networks, the selection of the centers is left as an open step, where they can be selected from the training set but can also be selected in other ways.

Example 10: Let us consider an example where we measure the similarity of images. It is unlikely to be very effective take the squared differences per pixel, and most images are likely to be dissimilar at the pixel level. Imagine our goal is to predict the sentiment for the image—positive, neutral or negative—as a three class classification problem. To do this well, it might be useful to know which objects are in the image, from a large collection of possible objects. Our measure of similarity could add up the number of similar objects between the two images, subtract the number of dissimilar objects and then normalize by the maximum number of objects in either image. Note that this is not a widely-used measure, it is just being used here to see that similarity can be defined in many different ways. □

Though we can use many different similarity functions, prototype representations often use *kernel* functions and so are also often called kernel representations. The function we used for RBF networks is actually a kernel, called the RBF kernel, $k(\mathbf{x}, \mathbf{c}_j) = \exp\left(-\frac{1}{2\sigma^2}\|\mathbf{x} - \mathbf{c}_j\|_2^2\right)$. More generally, there are a variety of different kernels including *linear* kernels $k(\mathbf{x}_1, \mathbf{x}_2) = \langle \mathbf{x}_1, \mathbf{x}_2 \rangle$ and *polynomial* kernels $k(\mathbf{x}_1, \mathbf{x}_2) = (\langle \mathbf{x}_1, \mathbf{x}_2 \rangle + a)^b$ for polynomial degree b. The defining characteristic of a kernel is that it can be written as a

dot-product in a transformed space: $k(\mathbf{x}_1, \mathbf{x}_2) = \langle \psi(\mathbf{x}_1), \psi(\mathbf{x}_2) \rangle$ where $\psi(\mathbf{x}_1)$ is a (typically) nonlinear function that transforms $\mathbf{x}_1$ to the new space.

You may wonder why we use these kernel functions to measure similarities, rather than just any arbitrary function. The reason is that this dot product form allows us to make strong theoretical claims about the properties of these functions. We discuss this briefly in the following advanced remark below, and also briefly revisit the utility of kernel functions in Section 15.3 when talking about Gaussian processes. This remark is labeled an advanced remark, because it goes to a level of theoretical knowledge not needed for this course, but is included to provide some justification for why these data representations have been considered worthwhile.

Advanced Remark 1: The key result that motivates the use of prototype representations with kernel similarities is the *representer theorem*. This theorem says the following. Assume our hypothesis space consists of functions $f : \mathcal{X} \to \mathbb{R}$ from a reproducing kernel Hilbert space (RKHS) $\mathcal{H}_k$ with associated kernel k. This means that the functions in this space have the form:

$$f(\mathbf{x}) = \sum_{j=1}^{p} w_j k(\mathbf{x}, \mathbf{c}_j)$$

for any $\mathbf{c}_1, \ldots, \mathbf{c}_p \in \mathcal{X}$, any $d \geq 1$ and weights $w_1, \ldots, w_p \in \mathbb{R}$. In other words, there are linear functions of these similarity features. Such functions may seem like a restricted space, but it actually encompasses many function classes. For example, we know it can represent certain classes of neural networks with ReLU activations, that we will discuss later. The representer theorem states that the function $f^* \in \mathcal{H}_k$ that minimizes error for a given dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, plus some regularization, is

$$f^*(\mathbf{x}) = \sum_{j=1}^{n} w_j^* k(\mathbf{x}, \mathbf{x}_j)$$

for some weights $w_1^*, \ldots, w_n^*$. In other words, it is a function that linearly weights features created by selecting the entire training dataset as the prototypes. This is true for any error, including all the errors we used for generalized linear models such as the squared error for linear regression and cross entropy for logistic regression. □

In practice, we will likely do not want to use the entire dataset, as n is likely very large. Instead, motivated by this result, we still use a function based on similarities to observed data, but only for a subset. Even with only a smaller set of prototypes, we still obtain a function from $\mathcal{H}_k$ and we can still obtain a function that is reasonably close to f^*. Of course, the quality of the approximation depends on how we select prototypes. This question has been thoroughly explored under the area of *active set selection*, but is beyond the scope of these notes. For a thorough set of references on the topic, see [23]. In the next section, we discuss one simple approach to subselect prototypes.

Example 11: One of the big advantages of prototype representations is that we can naturally handle a wider variety of input types. That is, we no longer require $\mathbf{x} \in \mathbb{R}^d$. Instead, as long as we can compute a similarity between the input vectors $\mathbf{x}_1$ and $\mathbf{x}_2$, then we can compute the prototype representation. Further, the resulting features given by the prototype representation *are* real-valued, making it easy to use our favorite method like linear regression or logistic regression.

For example, imagine the inputs consist of x_1 = full-time or part-time; x_2 = job type, out of possibilities { teacher, geologist, dancer }; and x_3 = age. Then we could use a matching similarity for the first two inputs—a similarity of 1 if it perfectly matches and 0 otherwise—and an RBF for age, since 32 is more similar to 31 than to 10. The resulting similarity between two instances $\mathbf{x}_1, \mathbf{x}_2$ would be

$$\begin{aligned} k(\mathbf{x}_1, \mathbf{x}_2) &= \mathbb{1}(x_{11} = x_{21})\,\mathbb{1}(x_{12} = x_{22})\,\text{RBF}(x_{13}, x_{23}) \\ &= \mathbb{1}(x_{11} = x_{21})\,\mathbb{1}(x_{12} = x_{22})\left(-\tfrac{1}{2\sigma^2}(x_{13} - x_{23})^2\right) \in [0, 1] \end{aligned}$$

To obtain the prototype representation, we would select p samples from the training set, either intelligently or randomly, for the prototypes $\mathbf{c}_1, \ldots, \mathbf{c}_p \in \mathcal{D}$. For any input $\mathbf{x}$, the prototype representation would consist of $\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^p$ the p similarities to these prototypical individuals selected from the training set. If we want to predict whether they are likely to get a disease, for example, then we would use logistic regression with this new representation, on a training set where we have recorded previous individuals and whether they contracted the disease. The learned weights outputted from logistic regression would be of size $\mathbf{w} \in \mathbb{R}^{p+1}$, because as usual we would add an intercept term. □

8.4 Feature Selection and Subselecting Prototypes

Once we used expand into a higher dimension, say with polynomial regression or prototype representations, it may be sensible to do *feature selection*. We likely overgenerate when we do this expansion, and we might want to remove low-utility features. We have already seen one mechanism to do so: adding ℓ_1 regularization to do feature selection discussed in Section 6.2. The ℓ_1 regularizer tries to find set weights to zero; for any weights that are zero, this is likely removing the feature.

For prototype representations, it has an additional interpretation: feature selection corresponds to prototype selection. Consider the following procedure. We create a prototype representation by using the entire training dataset as the prototypes, getting $\boldsymbol{\Phi} \in \mathbb{R}^{n \times p}$ with $p = n$. We then solve the ℓ_1 regularized optimization

$$c(\mathbf{w}) = \|\boldsymbol{\Phi}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1.$$

The ℓ_1 regularizer encourages elements in $\mathbf{w}$ to be zero. If $w_j = 0$, for example, then features $\boldsymbol{\phi}_j(\mathbf{x})$ is removed. Equivalently, it is like removing the j prototype. This optimization, therefore, should keep only the most important prototypes to accurately predict the targets on the training set.

The number of prototypes that are removed depends heavily on λ. If $\lambda = 0$, then the optimization has no incentive to remove any prototypes—no incentive to make any entries of $\mathbf{w}$ zero. It is always easier to fit the training data with more features. If λ is very big, it encourages using a very small number of prototypes; at the extreme, λ can be increased until the optimal choice is to set $\mathbf{w} = \mathbf{0}$. In between this extreme and 0, it is hard to predict how many prototypes will be selected.

Exercise 27: This objective is convex, and so we know we can obtain the global solution using our proximal method from Section 6.2. Imagine our optimization kept 10 prototypes. Does this mean we selected the best possible set of 10 prototypes for our training dataset?

To answer this, consider the objective we really want to optimize: $\|\mathbf{\Phi w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_0$ where the ℓ_0 regularizer simply counts non-zero entries: $\ell_0(\mathbf{w}) = \sum_{j=1}^{p} \mathbb{1}(w_j \neq 0)$. If we want to rewrite this to only select 10 prototypes, this would correspond to

$$\min_{\mathbf{w}\in\mathbb{R}^p, \|\mathbf{w}\|_0=10} \|\mathbf{\Phi w} - \mathbf{y}\|_2^2.$$

Do you think our ℓ_1 regularized solution will find the same prototypes? □

We can select λ either based on the desired number of prototypes or based on generalization performance. For example, if we know that we want to keep only 100 prototypes out of the $n = 10$k possible prototypes, then we might test a few values for λ until obtaining approximately 100 prototypes. Often, though, we are subselecting prototypes to improve generalization performance. In that case, we can leverage the internal cross-validation approach described in Section 7.4 to select a λ the reduces overfitting and improves generalization error.

Using ℓ_1 regularization for prototype selection can be seen as a simple form of representation learning. We allow prediction accuracy to inform which prototypes produces the most useful features. This approach incorporates a small amount of representation learning into otherwise fixed representations. In the next chapter, we talk about more flexible ways to learn representations.

Chapter 9

Learned Representations

In this chapter we discuss two common strategies to learn representations. We conclude the chapter showing that these two approaches have a common underlying theme, and are equivalent in one specific case. But, in general, they are otherwise quite distinct in that the algorithms and resulting representations are different.

9.1 Latent Factors and Factor Analysis

Imagine you are running a hospital and have information about your patients. You would like to get better at predicting whether you should use Strategy 1 or Strategy 2 to help ensure they follow the recommended treatment for a disease. You decide you will try to categorize patients according to the Big-Five personality test. This test identifies the level to which you are Open, Conscientious, Extraverted, Agreeable and Neurotic. You cannot directly measure these traits; rather, they are *latent* and have to be inferred based on the observed information you do have, such as past treatment behavior, self-identified preferences and health metrics.

This is the goal behind the broad class of methods that attempt to identify *latent factors*. Assume as usual that the input vector is a row vector $\mathbf{x} \in \mathbb{R}^{1\times d}$. In the simplest form, the idea is to find factors $\mathbf{h} \in \mathbb{R}^{1\times p}$ so that a linear weighting approximately produces the observed inputs

$$\mathbf{x} \approx \mathbf{h}\mathbf{D} = \sum_{k=1}^{p} h_k \mathbf{D}_{k,:}$$

where $\mathbf{D} \in \mathbb{R}^{p\times d}$ is called the *dictionary*, and also sometimes called the factor loading or coefficients. We use the term dictionary because each row $\mathbf{D}_{k,:} \in \mathbb{R}^d$ can be seen as a representative instance of an input that maximally has factor h_k. For example, if $\mathbf{h} = [0, 0, 1, 0, 0, \ldots, 0]$ with only $h_3 = 1$, then the resulting approximation of an input is exactly

$$\mathbf{h}\mathbf{D} = \sum_{k}^{p} h_k \mathbf{D}_{k,:} = \mathbf{D}_{3,:}.$$

Instead, if $\mathbf{h} = [0, 0.5, 0.5, 0, 0, \ldots, 0]$, then the approximation is 50% like dictionary item 2 and 50% like dictionary item 3.

Once you are given a dictionary, it is relatively straightforward to find the latent factors. For example, if our goal is to minimize the squared error when approximating the input $\tilde{\mathbf{x}}$, we can infer latent factors using

$$\tilde{\mathbf{h}} = \underset{\mathbf{h}\in\mathbb{R}^{1\times p}}{\operatorname{argmin}} \|\tilde{\mathbf{x}} - \mathbf{h}\mathbf{D}\|_2^2.$$

The $\tilde{\mathbf{h}}$ will weight the dictionary elements to best recreate $\tilde{\mathbf{x}}$.

Example 12: Consider again the example with the Big Five personality test. We assume we have $p = 5$ latent factors, one factor per personality component. Imagine we have 30 attributes per patient, so our inputs have dimension $d = 30$. Further imagine someone has already kindly found the dictionary for you. Now a new patient comes in, and fortunately you have the information about them that you need, namely you have the vector $\mathbf{x}$ of 30 attributes. Before you make any decisions about their care, you'd like to have a good idea where they stand on the Big Five.

To do so, you need to find $\mathbf{h}$ that is the optimal solution for $\|\mathbf{x} - \mathbf{hD}\|_2^2$. In fact, there is a closed-form solution for this

$$\begin{aligned}\nabla_{\mathbf{h}}\|\mathbf{x} - \mathbf{hD}\|_2^2 = (\mathbf{x} - \mathbf{hD})\mathbf{D}^\top &= \mathbf{0} \\ \implies \mathbf{xD}^\top - \mathbf{hDD}^\top &= 0 \\ \implies \mathbf{hDD}^\top &= \mathbf{xD}^\top \\ \implies \mathbf{h} &= \mathbf{xD}^\top(\mathbf{DD}^\top)^{-1}.\end{aligned}$$

This solution can have both positive and negative values in $\mathbf{h}$. For example, if $\mathbf{h} = [-0.6, 0.7, 0, 0.3, -0.1]$, then the patient is not very Open, is quite Conscientious, is neither Extraverted nor introverted, is somewhat agreeable and not really Neurotic.

We can similarly imagine what each dictionary element might mean. $\mathbf{D}_{1,:} \in \mathbb{R}^{30}$ might be key components of patient info for a patient that is very Open, $\mathbf{D}_{2,:} \in \mathbb{R}^{30}$ might be key components of patient info for a patient that is very Conscientious, and so on. When we reconstruct $\mathbf{x} = \mathbf{hD}$ we get $\mathbf{x} = -0.6\mathbf{D}_{1,:} + 0.7\mathbf{D}_{2,:} + 0.3\mathbf{D}_{4,:} - 0.1\mathbf{D}_{5,:}$. □

All of this assumes that we can actually find a $\mathbf{D}$ such that the factors corresponds to these five personality traits. In general, it is technically impossible to be sure that we have found an $\mathbf{h}$ and $\mathbf{D}$ that correspond perfectly to these five traits. This information is latent after all, and we can only do our best to infer it. If we had some ground-truth data, say from a psychologist measuring these properties for some number of patients, then we might have some hope to be confident that we found such a dictionary $\mathbf{D}$. But usually we do not have this extra labelling. The issue of extracting interpretable latent factors is an age-old problem in factor analysis. Fortunately for us in machine learning, our goal with data representations is usually to improve prediction accuracy, rather than answer scientific questions, and so we do not need to ensure latent factors match our interpretation. Our goal will simply be to find useful latent factors.

In the next two subsections we turn to the question of how we find this dictionary, so that we can identify these latent factors.

9.1.1 Matrix Factorization Approaches

Our goal is to find $\mathbf{D}$ such that $\mathbf{x}_i \approx \mathbf{h}_i\mathbf{D}$ for $i \in \{1, \ldots, n\}$. We can equivalently write this as $\mathbf{X} \approx \mathbf{HD}$ where $\mathbf{H} \in \mathbb{R}^{n \times p}$ has the ith row corresponding to $\mathbf{h}_i$ and $\mathbf{X}$ has $\mathbf{x}_i$ in the ith row. In other words, the data matrix $\mathbf{X}$ is factorized into a dictionary $\mathbf{D}$ and a basis or new representation $\mathbf{H}$ (see Figure 9.1).

In fact, many unsupervised learning algorithms (e.g., dimensionality reduction, sparse coding) and semi-supervised learning algorithms (e.g., supervised dictionary learning) can

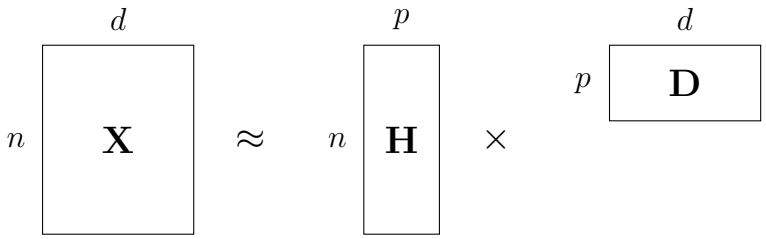


Figure 9.1: Matrix factorization of data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$. This factorization is a low-rank factorization, reducing the dimension from d to p, as would be given by PCA.

actually be formulated as matrix factorizations.[1] Unsupervised learning is about extracting underlying patterns, which is precisely what we are doing when we extract latent factors. The primary difference is that unsupervised learning can sometimes have a different goal, which is to visualize the data; for this goal, dimensionality reduction is key. This goal is still about data re-representation, but does not have a focus on predictive models or generative models. For visualization, for example, it might be worth considering only one or two factors to make it feasible to view the data. For prediction, we are unlikely to so severely restrict the model capacity.

Reducing the dimension with principal components analysis(PCA). PCA is a standard dimensionality reduction technique, where the input data $\mathbf{x} \in \mathbb{R}^{1 \times d}$ is projected into a lower dimensional $\mathbf{h} \in \mathbb{R}^{1 \times p}$ spanned by the space of principal components. These principal components are the directions of maximal variance in the data. To obtain these p principal components $\mathbf{D} \in \mathbb{R}^{p \times d}$, the common solution technique is to obtain the singular value decomposition of the data matrix $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \in \mathbb{R}^{n \times d}$, giving

$$\mathbf{D} = \mathbf{V}_p^\top \in \mathbb{R}^{p \times d}$$
$$\mathbf{H} = \mathbf{U}_p \mathbf{\Sigma}_p \in \mathbb{R}^{n \times p}$$

where $\mathbf{\Sigma}_p \in \mathbb{R}^{p \times p}$ consists of the top largest p singular values (in descending order) and $\mathbf{U}_p \in \mathbb{R}^{n \times p}$ and $\mathbf{V}_p \in \mathbb{R}^{p \times d}$ are the corresponding singular vectors, i.e., $\mathbf{U}_p = \mathbf{U}_{:,1:p}$ and $\mathbf{V}_p = \mathbf{V}_{:,1:p}$. The new representation for $\mathbf{X}$ (using PCA) is this $\mathbf{H}$. Note that PCA does not subselect features, but rather creates new features: the generated $\mathbf{h}$ is not a subset of the original $\mathbf{x}$.

This dimensionality reduction technique can also be formulated as a matrix factorization. The corresponding optimization has been shown to be

$$\min_{\mathbf{D} \in \mathbb{R}^{p \times d}, \mathbf{H} \in \mathbb{R}^{n \times p}} \|\mathbf{X} - \mathbf{H}\mathbf{D}\|_F^2$$

The rank p matrix $\hat{\mathbf{X}}$ that best approximates $\mathbf{X}$, in terms of minimal Frobenius norm, is $\hat{\mathbf{X}} = \mathbf{U}_p \mathbf{\Sigma}_p \mathbf{V}_p^\top$. You do not need to be able to solve this optimization problem, but this result is actually quite simply to derive[2], and we can give the intuition here. The Frobenius norm is invariant to orthonormal matrices and so we can write it as just the sum of the

[1] For a thorough overview of these connections, see [30].

[2] If you are interested in the formal proof, see the Eckart-Young-Mirsky theorem.

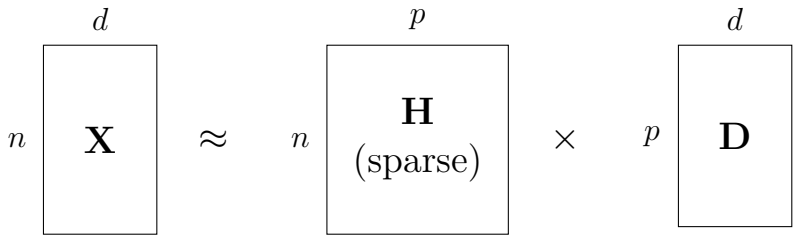


Figure 9.2: Matrix factorization of data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ for sparse coding. The matrix $\mathbf{H}$ is constrained to be sparse, using ℓ_1 regularization. If it was not constrained to be sparse, then this factorization would result in a trivial solution. In other words, we could set $\mathbf{H} = [\mathbf{X}\ \mathbf{0}]$ and $\mathbf{D} = \mathbf{I}$ and perfectly recover $\mathbf{X}$.

squared singular values, $\|\mathbf{X}\|_F^2 = \sum_{j=1}^{d} \sigma_j^2$. This means that the rank p matrix $\hat{\mathbf{X}}$ that minimizes $\|\mathbf{X} - \hat{\mathbf{X}}\|_F^2$ is the one that matches the largest singular values, incurring the smallest error that is possible for a p rank matrix: $\|\mathbf{X} - \hat{\mathbf{X}}\|_F^2 = \sum_{j=p+1}^{d} \sigma_j^2$. The sum of squared errors for the smallest singular values gives us the lowest error. Once we know $\hat{\mathbf{X}} = \mathbf{U}_p \mathbf{\Sigma}_p \mathbf{V}_p^\top$ is the minimizer, then we can simply assign $\mathbf{H} = \mathbf{U}_p \mathbf{\Sigma}_p$ to first part of this decomposition and $\mathbf{D} = \mathbf{V}_p^\top$ to the second part, to get $\hat{\mathbf{X}} = \mathbf{HD}$.

The primary reason that $\mathbf{h}$ generated by PCA could be useful as a representation is to prevent overfitting. The projection to lower dimensions has the property that it removes noise and maintains only the most meaningful directions. This projection reduces the number of features and promotes generalization, by preventing overfitting to the noise. This is beneficial if there are a large number of inputs, either because of high-dimensional data or because the data was first augmented using polynomials or a prototype representation.

Exercise 28: PCA is also often described by talking about the eigenvalue decomposition of $\mathbf{X}^\top \mathbf{X}$ rather than the singular value decomposition of $\mathbf{X}$. Further, for PCA, we often say that the principal components project our input data to the lower-dimensional space. Verify that the top p eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_p$ correspond to the squared values of the top singular values $\sigma_1^2, \sigma_2^2, \ldots, \sigma_p^2$. Further verify that the principal components $\mathbf{D} = \mathbf{V}_p^\top$ are the top p eigenvectors of $\mathbf{X}^\top \mathbf{X}$. Finally, show that multiplying by these top eigenvector (principal components) projects $\mathbf{X}$ to our new representation $\mathbf{H}$, i.e., show $\mathbf{XD}^\top = \mathbf{H}$. *Hint:* Some of this derivation is given to you in Section 9.3. □

Increasing the dimension with sparse coding. Sparse coding is biologically motivated [18], based on sparse activations for memory in the mammalian brain. In this case, $p >> d$ and the idea is that only a small set of dictionary items are used per input. We have a very large dictionary of representative attributes from which to select: $\mathbf{D} \in \mathbb{R}^{p \times d}$ is large and dense, with many rows. A small number of these are linearly combined to produce the input: for an input $\mathbf{x}$ the $\mathbf{h} \in \mathbb{R}^p$ is sparse with only a few non-zero entries, to create $\mathbf{x} \approx \mathbf{hD}$. This factorization is visualized in Figure 9.2.

Example 13: Let us first consider an example of how using a higher-dimensional sparse representation might be more suitable than a low-dimensional representation. Sparse coding has been used for image representations, shown in Figure 9.3, for vision. Each dictionary item might corresponds to a basic edge type, for example, or typical shapes found in image

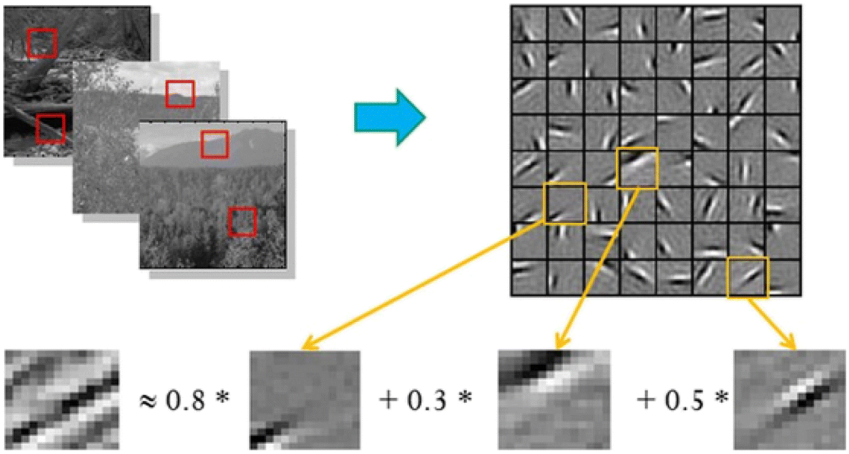


Figure 9.3: The dictionary found with sparse coding, for recreating small patches in an image. Image from Andrew Ng and his lab, in 2015.

patches. Linearly combining each of these produces more complex image patches. Sparse coding allows for a large collection of such types to be found, and only a very small subset to be used per each patch in the image. Together, the new representation for the entire image is a stacking of the encodings for each patch, which itself still only uses a very small subset of possible dictionary elements and is a sparse representation of the image.

This re-representation has been used for denoising or even sharpening an image. By reconstructing the image using the dictionary, we can replace each image patch using denoised, canonical shapes. If a patch was blurry or noisy, it can be reconstructed using the closest collection of (sharp, non-blurry) dictionary elements.

Once we have this representation, it should also be useful for classification. Sparse coding extracts the key shapes per image patch, giving us a lot of information about the image. It abstracts from the pixel level to a shape or object level. Classification is ultimately about grouping similar inputs into classes. These features are a should allow us to identify that two images have similar sets of shapes/objects, and so should be classified similarly. □

To get the sparse representation in sparse coding, we again leverage a regularizer that sets values to zero: the ℓ_1. Specifically, for a given $\mathbf{x}$, we find a new $\mathbf{h}$ with the optimization $\min_{\mathbf{h}\in\mathbb{R}^p} \|\mathbf{x} - \mathbf{hD}\|_2^2 + \lambda\|\mathbf{h}\|_1$. This objective encourages as many values of $\mathbf{h}$ to be zero as possible, with the level of sparsity dictated by λ and the size of the new dimension p. Solving jointly for the $\mathbf{H}$ and $\mathbf{D}$ for sparse coding is a bit more complicated than PCA, so we do not go into the algorithm more deeply here. Our primary goal here is to understand the types of representations we can learn using a matrix factorization, and why their properties might be beneficial. If you are interested in the algorithms to learn these sparse representations with sparse coding, then see Appendix A.5.1.

9.1.2 Probabilistic Approaches

Now let us do this all again, but this time be explicit about our probabilistic assumptions. We assume first that there is some distribution over latent variables, $p(\mathbf{h})$. Specifically,

we make a Gaussian assumption $p(\mathbf{h}) = \mathcal{N}(\boldsymbol{\mu} = \mathbf{0}, \mathbf{I})$, reflecting that most latent variables will be around zero, with decreasing probability that they are very large values far away from zero. Additionally, we model that there is not perfect certainty which latent variables produced $\mathbf{x}$, or equivalently, that each $\mathbf{h}$ could produce different $\mathbf{x}$. We model this by assuming $p(\mathbf{x}|\mathbf{h}, \mathbf{D}) = \mathcal{N}(\boldsymbol{\mu} = \mathbf{hD}, \sigma^2\mathbf{I})$ for some $\sigma > 0$. This assumption looks a lot like our regression assumption, where we model the expectation as a linear function, $\mathbb{E}[\boldsymbol{X}|\mathbf{h}] = \mathbf{hD}$, with some noise around that mean to account for the fact that we do not have a deterministic relationship between $\mathbf{hD}$ and $\mathbf{x}$. We know that there will be some error in reconstructing $\mathbf{x}$ with $\mathbf{hD}$, and this noise reflects that error.

Our learning goal is to find $\mathbf{D}$ under these assumptions, where

$$p(\mathbf{x}|\mathbf{D}) = \int p(\mathbf{x}|\mathbf{h}, \mathbf{D})p(\mathbf{h})d\mathbf{h}$$

As usual, we want to maximum the likelihood of the data, so find the dictionary that makes the data the most likely under the above Gaussian assumptions. Though we skip the steps here, if we plug in the Gaussian assumptions above, we can get a closed-form solution to this integral

$$p(\mathbf{x}|\mathbf{D}) = \int p(\mathbf{x}|\mathbf{h}, \mathbf{D})p(\mathbf{h})d\mathbf{h} = \mathcal{N}(\boldsymbol{\mu} = \mathbf{0}, \mathbf{D}^\top\mathbf{D} + \sigma^2\mathbf{I})$$

We can see that our assumptions imply that $\mathbf{x}$ is centered—mean zero. The formalism can easily be extended to allow a non-zero mean (see [2, Section 21.1]), but to understand the idea more simply here, we assume $\mathbf{x}$ is centered. Further, we can see that it assumes $\mathbf{x}$ primarily lies in a lower-dimensional space. The term $\mathbf{D}^\top\mathbf{D} \in \mathbb{R}^{d\times d}$ is low-rank, because $p < d$. The covariance $\mathbf{D}^\top\mathbf{D} + \sigma^2\mathbf{I}$, therefore, implies $\mathbf{x}$ is primarily in this low-dimensional space with only small noise from σ^2 making it minorly deviate from this plane.

The maximum likelihood estimator for $\mathbf{D}$ and σ can be obtained using the SVD of $\mathbf{X}$ with singular values σ_j. Again, we skip the steps, since the outcome is what we are after here; see [2, Section 21] for these details.

$$\sigma^2_{\text{MLE}} = \frac{1}{d-p}\sum_{j=p+1}^{d}\sigma_j^2$$
$$\mathbf{D}_{\text{MLE}} = (\boldsymbol{\Sigma}_p^2 - \sigma^2_{\text{MLE}}\mathbf{I})^{1/2}\mathbf{V}_p^\top$$

Notice the similarity to the PCA solution. The primary difference is that the singular values are shifted downwards by σ_{MLE}. The other difference is superficial, which relates to whether we put the singular values in $\mathbf{H}$ or $\mathbf{D}$. By convention, $\mathbf{D}$ for PCA is set to just the singular vectors, resulting in solution $\mathbf{D} = \mathbf{V}_p^\top$ and $\mathbf{H} = \mathbf{U}_p\Sigma_p$. However, a perfectly equivalent solution is $\mathbf{D} = \Sigma_p^{1/2}\mathbf{V}_p^\top$ and $\mathbf{U}_p\Sigma_p^{1/2}$ because the product $\mathbf{HD}$ is exactly the same in both cases. This convention is the one used in probabilistic PCA.

The other difference is in how we might extract a representation $\mathbf{h}$ for input $\mathbf{x}$. If we opt to find the $\mathbf{h}$ that makes the input the most likely, then we actually get the same solution as for matrix factorization

$$\operatorname*{argmin}_{\mathbf{h}\in\mathbb{R}^p} -\ln p(\mathbf{x}|\mathbf{h}, \mathbf{D}) = \operatorname*{argmin}_{\mathbf{h}\in\mathbb{R}^p} \|\mathbf{x} - \mathbf{hD}\|_2^2$$

If, on the other hand, we would like to get the most likely $\mathbf{h}$ for a given input, we optimize

$$\begin{aligned}\underset{\mathbf{h}\in\mathbb{R}^p}{\operatorname{argmin}} -\ln p(\mathbf{h}|\mathbf{x},\mathbf{D}) &= \underset{\mathbf{h}\in\mathbb{R}^p}{\operatorname{argmin}} -\ln p(\mathbf{x}|\mathbf{h},\mathbf{D}) - \ln p(\mathbf{h}) \\ &= \underset{\mathbf{h}\in\mathbb{R}^p}{\operatorname{argmin}} \|\mathbf{x}-\mathbf{h}\mathbf{D}\|_2^2 + \sigma_{\text{MLE}}^2\|\mathbf{h}\|_2^2\end{aligned}$$

where the last line follows from the unit variance assumption on $\mathbf{h}$ and the fact that the variance of $\mathbf{x}$ given $\mathbf{h}$ is $\sigma_{\text{MLE}}^2\mathbf{I}$. This objective is somewhat more sensible, since it matches our assumptions when training $\mathbf{h}$.

When we move beyond Gaussian assumptions with diagonal variance, we rarely obtain these nice closed-form solutions. Even for non-diagonal variances with Gaussian assumptions, an iterative algorithm is required, called Factor Analysis. The same concepts about identifying latent factors, and how to think about $\mathbf{h}$ and $\mathbf{D}$, continue to apply, even though the algorithms can become a bit more complex. We will discuss this further, in Chapter **??**.

9.2 Learning Representations with Neural Networks

Neural networks allows us to obtain nonlinear transformations of the data. The addition of hidden layers, with non-linear activation functions, enables learning of nonlinear functions f of inputs to produce predictions of targets. In this section, we first explain how this nonlinear function is constructed, then how we pick the loss function and finally the algorithm to find the parameters for this function.

9.2.1 Functions Produced by a Neural Network

Let us start with an intuitive example of the types of function obtained with neural networks. Figure 9.4 shows the graphical model for the generalized linear models we discussed in the previous chapters, where the weights and corresponding transfer can be thought of as being on the arrows (as they are not random variables). Figure 9.5 shows a neural network with one hidden-layer; this is called a two-layer neural network, as there are two layers of weights. In the figure, the neural network inputs a 4-dimensional feature vector $\mathbf{x} = [x_1,\ x_2,\ x_3,\ x_4]$ (i.e., $d = 4$) and outputs a 2-dimensional prediction $\mathbf{y} = [y_1,\ y_2]$ (i.e., $m = 2$). The hidden layer consists of a mapping from $\mathbf{x}$ to a new representation that is 5-dimensional (i.e., $p_1 = 5$ as per the notation below). For the neural network, let each node in this hidden representation be indexed by $k \in \{1,\ldots,5\}$. Each h_k consists of a transformation of a linear weighting of $\mathbf{x}$, such as a sigmoid activation: $h_k = \sigma\left(\sum_{j=1}^d x_j w_{kj}\right) = \sigma\left(\mathbf{x}\mathbf{w}_k\right)$ where $\mathbf{w}_k \in \mathbb{R}^d$ is the weights on the first layer used to produce the kth node in the hidden representation.

Example 14: Let us continue this example, but make it even simpler by considering $d = 1$ (i.e., one input observation), $m = 1$ (i.e., one output), $p_1 = 2$ (i.e., 2-dimensional hidden layer) and a sigmoid activation to get the first hidden layer. Assume we are given one instance (x, y). Then input observation x is transformed into

$$\mathbf{h} = [h_1,\ h_2], \qquad \text{with } h_1 = \sigma(xw_1^{(2)}) \text{ and } h_2 = \sigma(xw_2^{(2)}) \qquad \text{for } w_1^{(2)}, w_2^{(2)} \in \mathbb{R}.$$

We use the superscript notation to distinguish between the weights in the first and last layer. It may seem backwards that we label $\mathbf{w}^{(2)}$ for the input layer and $\mathbf{w}^{(1)}$ for the output

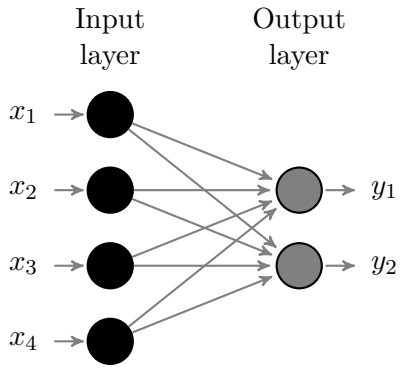


Figure 9.4: Generalized linear model, such as logistic regression.

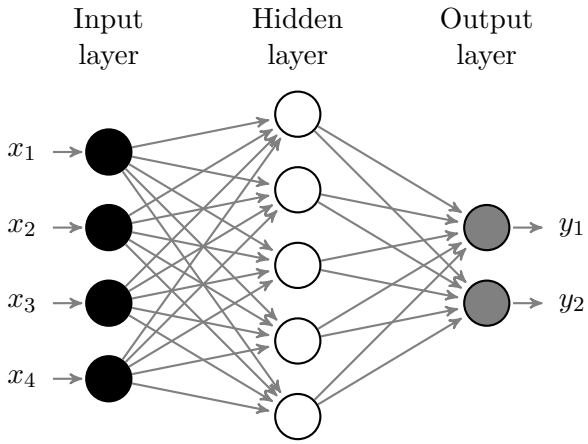


Figure 9.5: Standard two-layer neural network.

layer, but you will see below that it makes notation simpler to start indexing from the output layer.

Once we have $\mathbf{h}$, we can pretend that $\mathbf{h}$ is the new input representation and go ahead and learn a (generalized) linear model on this last layer. Let's consider two cases: $y \in \mathbb{R}$ and $y \in \{0, 1\}$. If $y \in \mathbb{R}$, we use linear regression for this last layer and so learn weights $\mathbf{w}^{(1)} \in \mathbb{R}^2$ such that $\mathbf{h}\mathbf{w}^{(1)}$ approximates the true output y. If $y \in \{0, 1\}$, we use logistic regression for this last layer and so learn weights $\mathbf{w}^{(1)} \in \mathbb{R}^2$ such that $\sigma(\mathbf{h}\mathbf{w}^{(1)})$ approximates the true output y. □

Now we consider the more general case with any d, p_1, m. Let's assume we are doing regression for simplicity. For linear regression we estimated $\mathbf{W} \in \mathbb{R}^{d \times m}$ to get function $f(\mathbf{x}) = \mathbf{x}\mathbf{W} \approx \mathbf{y}$. When we add a hidden layer, we have two parameter matrices $\mathbf{W}^{(2)} \in \mathbb{R}^{d \times p_1}$ and $\mathbf{W}^{(1)} \in \mathbb{R}^{p_1 \times m}$, where p_1 is the dimension of the hidden layer

$$\mathbf{h} = \sigma(\mathbf{x}\mathbf{W}^{(2)}) = \begin{bmatrix} \sigma(\mathbf{x}\mathbf{W}^{(2)}_{:1}) \\ \sigma(\mathbf{x}\mathbf{W}^{(2)}_{:2}) \\ \vdots \\ \sigma(\mathbf{x}\mathbf{W}^{(2)}_{:p_1}) \end{bmatrix}^\top \in \mathbb{R}^{1 \times p_1}$$

where the sigmoid function is applied to each entry in $\mathbf{x}\mathbf{W}^{(2)}$. This hidden layer is the new set of features and again you will do the regular linear regression optimization to learn weights on $\mathbf{h}$:

$$\mathbb{E}[Y|\mathbf{x}] \approx \mathbf{h}\mathbf{W}^{(1)} = \sigma(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)}.$$

Intuitively, $\sigma(\mathbf{x}\mathbf{W}^{(2)})$ is our new representation of the inputs: it is a nonlinear transformation. We then learn a linear function on this new representation, meaning that we learn a nonlinear function in terms of the original inputs $\mathbf{x}$.

We can apply this linear+activation transformation for multiple layers, resulting in a nested (deep) transformation. Denote each differentiable activation function $f_1, \ldots, f_H$, ordered with f_1 as the output activation, and $p_1, \ldots, p_{H-1}$ as the hidden dimensions with

$H-1$ hidden layers. Then the output from the neural network is

$$f_1\left(f_2\left(\dots f_{H-1}\left(f_H\left(\mathbf{x}\mathbf{W}^{(H)}\right)\mathbf{W}^{(H-1)}\right)\dots\right)\mathbf{W}^{(1)}\right)$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{p_1 \times m}$, $\mathbf{W}^{(2)} \in \mathbb{R}^{p_2 \times p_1}$, $\dots$, $\mathbf{W}^{(H)} \in \mathbb{R}^{d \times p_{H-1}}$.

9.2.2 Activations and Loss Functions

The next question is how we select the loss functions for the prediction and activations for our network. There is at least a relatively straightforward answer to how we specify the loss and output activation f_1: the same choice as we made for (generalized) linear models. If we are doing logistic regression, then we use the cross-entropy loss with $f_1 = \sigma$. If we are doing multinomial logistic regression, then we use the corresponding loss with $f_1 = \text{softmax}$. If we are doing regression, then we use the squared-error with $f_1 = \text{identity}$. We first decide the distribution for $p(y|\mathbf{x})$, and the resulting maximum likelihood problem gives us the choice of f_1 and loss based on the GLM formulation.

The primary difference to GLMs is that now the parameters for the GLM are a nonlinear function of $\mathbf{x}$. Previously in GLMs we had that $\mathbb{E}[Y|\mathbf{x}] \approx g(\mathbf{x}\mathbf{W})$. Now, once we first use a transformation with neural networks, we have that $\mathbb{E}[Y|\mathbf{x}] \approx g(\mathbf{h}\mathbf{W}^{(1)})$ where $\mathbf{h} = f_2\left(\dots f_{H-1}\left(f_H\left(\mathbf{x}\mathbf{W}^{(H)}\right)\mathbf{W}^{(H-1)}\right)\dots\right)$ and the output activation f_1 is the transfer g. For example, previously in linear regression we assumed that $p(y|\mathbf{x}) = \mathcal{N}(\mathbf{x}\mathbf{W}, \sigma^2)$. Now we assume that the mean can be a nonlinear function of $\mathbf{x}$, specifically $\mathbf{h}\mathbf{W}^{(1)}$.

The transfer at the end of the network is determined by the loss, but the activations *within* the network are not as clear-cut. This choice simply produces different nonlinear functions. Typical options include the rectified linear unit (ReLU), tanh and sigmoid. ReLU is a common default, explained more below. Note though that many functions could be used, and the key criteria are that it is (a) differentiable and (b) does not make optimization too difficult. Ultimately, we will take the gradients of these functions, and so we want the activations to be differentiable. And some activations seem to result in a more complex optimization surface, causing practitioners to gravitate towards activations that make gradient descent perform better. It is a ripe area for exploration to better understand activations inside a network, and I have no doubt that we will find better choices in the next few year. Today, though, ReLU remains a typical default.

ReLU is a simple thresholding function: for a given scalar θ, it returns $\max(0, \theta)$. If we use ReLU for a network with two hidden layers ($f_2 = f_3 = \text{ReLU}$) and a prediction for regression ($f_1 = \text{identity}$), then we would have

$$f(\mathbf{x}) = f_1\left(f_2\left(f_3\left(\mathbf{x}\mathbf{W}^{(3)}\right)\mathbf{W}^{(2)}\right)\mathbf{W}^{(1)}\right) = \max\left(0, \max\left(0, \mathbf{x}\mathbf{W}^{(3)}\right)\mathbf{W}^{(2)}\right)\mathbf{W}^{(1)}.$$

The innermost term is a vector $\boldsymbol{\theta}^{(3)} \stackrel{\text{def}}{=} \mathbf{x}\mathbf{W}^{(3)} \in \mathbb{R}^{p_2}$. The ReLU is applied elementwise to each entry in $\boldsymbol{\theta}^{(3)}$, to produce the hidden layer

$$\mathbf{h}^{(2)} \stackrel{\text{def}}{=} f_3(\boldsymbol{\theta}^{(3)}) = \max(0, \boldsymbol{\theta}^{(3)}).$$

Similarly, we have $\boldsymbol{\theta}^{(2)} \stackrel{\text{def}}{=} \mathbf{h}^{(2)}\mathbf{W}^{(2)} \in \mathbb{R}^{p_1}$ and $\mathbf{h}^{(1)} \stackrel{\text{def}}{=} f_2(\boldsymbol{\theta}^{(2)}) = \max(0, \boldsymbol{\theta}^{(2)})$. Finally the

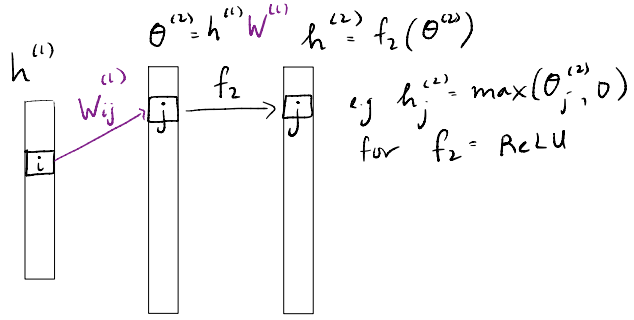


Figure 9.6: The terms in one layer of a neural network.

output is $\boldsymbol{\theta}^{(1)} \stackrel{\text{def}}{=} \mathbf{h}^{(1)}\mathbf{W}^{(1)} \in \mathbb{R}^m$.

$$\begin{aligned} f(\mathbf{x}) &= \max\left(0, \max\left(0, \mathbf{x}\mathbf{W}^{(3)}\right)\mathbf{W}^{(2)}\right)\mathbf{W}^{(1)} \\ &= \max\left(0, \max\left(0, \boldsymbol{\theta}^{(3)}\right)\mathbf{W}^{(2)}\right)\mathbf{W}^{(1)} \\ &= \max\left(0, \mathbf{h}^{(2)}\mathbf{W}^{(2)}\right)\mathbf{W}^{(1)} \\ &= \max\left(0, \boldsymbol{\theta}^{(2)}\right)\mathbf{W}^{(1)} \\ &= \mathbf{h}^{(1)}\mathbf{W}^{(1)} \end{aligned}$$

We visualize each of these terms in Figure 9.6.

9.2.3 The Backpropagation Algorithm

The backpropagation algorithm is simply gradient descent on the loss for the neural network, with a careful ordering of computation to avoid repeating computation. In particular, one first propagates forward and computes variable $\mathbf{h} = f_2(\mathbf{x}\mathbf{W}^{(2)}) \in \mathbb{R}^{1\times p}$ and then $\hat{\mathbf{y}} = f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)}) = f_1(\mathbf{h}\mathbf{W}^{(1)})$. We then compute the error between our prediction $\hat{\mathbf{y}}$ and the true label. We take the gradient of this error (loss) w.r.t. to our parameters. For efficient computation, the best ordering is to compute the gradient w.r.t. to the last parameter $\mathbf{W}^{(1)}$ first, and then $\mathbf{W}^{(2)}$. This is the reason for the term backpropagation, since the error is propagated backward from the last layer first.

Let us first go through an example with the cross-entropy loss and sigmoid activation within the network, for a two-layer network. We compute this gradient assuming we only have one sample $(\mathbf{x}, \mathbf{y})$, since that automatically gives us the gradient for a mini-batch (or even the full batch). That is, we sum these gradients for each individual sample over the mini-batch. Our goal is to compute the partial derivative for each weight w.r.t. the loss $c(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}) = \ell(y, \hat{y}) = -y \ln \hat{y} - (1-y)\ln(1-\hat{y})$ where $\hat{y}$ is produced using the neural network with weights $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}$.

First, we take the partial derivative w.r.t. the parameters $\mathbf{W}^{(1)} \in \mathbb{R}^{p_1}$. Notice that the

output of the network is $\hat{y} = \sigma(\boldsymbol{\theta}^{(1)}) = \sigma(\mathbf{h}^{(1)}\mathbf{W}^{(1)})$.

$$\begin{aligned}
\frac{\partial c(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}_j^{(1)}} &= \frac{\partial \ell(\hat{y}, y)}{\partial \mathbf{W}_j^{(1)}} \\
&= \frac{\partial \ell(\hat{y}, y)}{\partial \boldsymbol{\theta}^{(1)}} \frac{\partial \boldsymbol{\theta}^{(1)}}{\partial \mathbf{W}_j^{(1)}} \\
&= (\sigma(\mathbf{h}^{(1)}\mathbf{W}^{(1)}) - y) \frac{\partial \boldsymbol{\theta}^{(1)}}{\partial \mathbf{W}_j^{(1)}} \\
&= (\sigma(\mathbf{h}^{(1)}\mathbf{W}^{(1)}) - y)\mathbf{h}_j^{(1)}
\end{aligned}$$

where the last line follows from the fact that $\frac{\partial \boldsymbol{\theta}^{(1)}}{\partial \mathbf{W}_j^{(1)}} = \frac{\partial \mathbf{h}^{(1)}\mathbf{W}^{(1)}}{\partial \mathbf{W}_j^{(1)}} = \frac{\partial \sum_{i=1}^{p_1} \mathbf{h}_i^{(1)}\mathbf{W}_i^{(1)}}{\partial \mathbf{W}_j^{(1)}} = \mathbf{h}_j^{(1)}$. This derivation comes from noticing that the last layer of the network can be thought of as a GLM with inputs $\mathbf{h}^{(1)}$. Therefore, we can simply re-use the derivation we used for the logistic regression update, with input $\mathbf{h}^{(1)}$. In particular, we re-use the fact that for $\hat{y} = \sigma(\theta)$, we have $\frac{\partial \ell(\hat{y},y)}{\partial \theta} = (\sigma(\theta) - y)$.

Exercise 29: Verify the above update by deriving the partial derivative $\frac{\partial \ell(\hat{y},y)}{\partial \mathbf{W}_j^{(1)}}$, by doing it yourself from scratch without using the GLM update. □

Next, we compute the partial derivative with respect to $\mathbf{W}^{(2)} \in \mathbb{R}^{d \times p_1}$. We will use the chain rule to reuse part of the above update, by defining

$$\delta^{(1)} \stackrel{\text{def}}{=} (\sigma(\mathbf{h}^{(1)}\mathbf{W}^{(1)}) - y)$$

and consider how the objective changes when we change the entry $\mathbf{W}_{ij}^{(2)}$ at the ith row and jth column of $\mathbf{W}^{(2)}$.

$$\begin{aligned}
\frac{\partial c(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} &= \frac{\partial \ell(\hat{y}, y)}{\partial \mathbf{W}_{ij}^{(2)}} \\
&= \frac{\partial \ell(\hat{y}, y)}{\partial \boldsymbol{\theta}^{(1)}} \frac{\partial \boldsymbol{\theta}^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} \\
&= \delta^{(1)} \frac{\partial \sum_{k=1}^{p_1} \mathbf{h}_k^{(1)}\mathbf{W}_k^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} \\
&= \delta^{(1)} \sum_{k=1}^{p_1} \mathbf{W}_k^{(1)} \frac{\partial \mathbf{h}_k^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}}
\end{aligned}$$

Recall that $\mathbf{h}_k^{(1)} = f_2(\boldsymbol{\theta}_k^{(2)})$ with $\boldsymbol{\theta}_k^{(2)} = \mathbf{x}\mathbf{W}_{:k}^{(2)} = \sum_{i=1}^{p_1} \mathbf{x}_i \mathbf{W}_{ik}^{(2)}$, and so

$$\frac{\partial \mathbf{h}_k^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} = \frac{\partial f_2(\boldsymbol{\theta}_k^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} = \frac{\partial f_2(\boldsymbol{\theta}_k^{(2)})}{\partial \boldsymbol{\theta}_k^{(2)}} \frac{\partial \boldsymbol{\theta}_k^{(2)}}{\partial \mathbf{W}_{ij}^{(2)}} \quad \text{where } \frac{\partial \boldsymbol{\theta}_k^{(2)}}{\partial \mathbf{W}_{ij}^{(2)}} = \begin{cases} 0 & \text{if } k \neq j \\ x_i & \text{if } k = j \end{cases}$$

The two cases arise from the fact that $\mathbf{W}_{ij}^{(2)}$ only influences the jth hidden node: it connects the ith input to the jth hidden node. Since it does not influence the kth hidden node when $k \neq j$, the partial derivative for $\boldsymbol{\theta}_k^{(2)}$ for $k \neq j$ is zero.

Putting this back together, we get that

$$\sum_{k=1}^{p_1} \mathbf{W}_k^{(1)} \frac{\partial \mathbf{h}_k^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} = \mathbf{W}_j^{(1)} \frac{f_2(\boldsymbol{\theta}_k^{(2)})}{\partial \boldsymbol{\theta}_k^{(2)}} x_i$$

When $f_2 = \sigma$, we can use the fact that

$$\frac{\partial \sigma(\theta)}{\partial \theta} = (1 - \sigma(\theta))\sigma(\theta).$$

We can define the new error (delta) that is propagating back to $\mathbf{W}_{ij}^{(2)}$ as

$$\delta_j^{(2)} \stackrel{\text{def}}{=} \delta^{(1)} \mathbf{W}_j^{(1)} (1 - \sigma(\boldsymbol{\theta}_j^{(2)}))\sigma(\boldsymbol{\theta}_j^{(2)})$$

and get partial derivative

$$\frac{\partial c(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} = \delta_j^{(2)} x_i.$$

Intuitively, $\delta_j^{(2)}$ includes the error on the output ($\delta^{(1)}$), down-weighted by the strength of the connection $\mathbf{W}_j^{(1)}$ for the jth hidden node to the output. We can think of this as the error travelling back from the output—or propagating backwards—along the connections. Additionally, the error is modulated by how much changing $\mathbf{W}_{ij}^{(2)}$ could have changed the activation, namely by the derivative of $\sigma(\boldsymbol{\theta}_j^{(2)})$.

We can generalize these update rules to more hidden layers and to multiple outputs. If we have two hidden layers, and a vector of outputs $m > 1$, then we have weights $\mathbf{W}^{(1)} \in \mathbb{R}^{p_1 \times m}, \mathbf{W}^{(2)} \in \mathbb{R}^{p_2 \times p_1}, \mathbf{W}^{(3)} \in \mathbb{R}^{d \times p_2}$ and activations f_3, f_2. Let f_2' and f_3' be the derivatives of these activations. For the sigmoid, we saw it was $\sigma'(\theta) = \sigma(\theta)(1 - \sigma(\theta))$, for ReLu it is $\text{ReLu}'(\theta) = \mathbb{1}(\theta > 0)$ (0 if $\theta \leq 0$ and 1 if $\theta > 0$). We first do a **forward pass** for an input $\mathbf{x}$, to get variables

$$\begin{aligned}
\boldsymbol{\theta}^{(3)} &= \mathbf{x}\mathbf{W}^{(3)} \\
\mathbf{h}^{(2)} &= f_3(\boldsymbol{\theta}^{(3)}) \\
\boldsymbol{\theta}^{(2)} &= \mathbf{h}^{(2)}\mathbf{W}^{(2)} \\
\mathbf{h}^{(1)} &= f_2(\boldsymbol{\theta}^{(2)}) \\
\boldsymbol{\theta}^{(1)} &= \mathbf{h}^{(1)}\mathbf{W}^{(1)} \\
\hat{\mathbf{y}} &= f_1(\boldsymbol{\theta}^{(1)})
\end{aligned}$$

Then, to compute the **gradient descent (aka backpropagation)** updates, we use

$$\begin{aligned}
\boldsymbol{\delta}^{(1)} &= \hat{\mathbf{y}} - \mathbf{y} \\
\boldsymbol{\delta}_j^{(2)} &= \left(\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)}\right) f_2'(\boldsymbol{\theta}_j^{(2)}) \quad \text{for all } j \in \{1, \ldots, p_1\} \\
\mathbf{W}_{ij}^{(1)} &\leftarrow \mathbf{W}_{ij}^{(1)} - \eta \boldsymbol{\delta}_j^{(1)} \mathbf{h}_i^{(1)} \quad \text{for all } i \in \{1, \ldots, p_1\}, j \in \{1, \ldots, m\} \\
\boldsymbol{\delta}_j^{(3)} &= \left(\mathbf{W}_{j:}^{(2)} \boldsymbol{\delta}^{(2)}\right) f_3'(\boldsymbol{\theta}_j^{(3)}) \\
\mathbf{W}_{ij}^{(2)} &\leftarrow \mathbf{W}_{ij}^{(2)} - \eta \boldsymbol{\delta}_j^{(2)} \mathbf{h}_i^{(2)} \quad \text{for all } i \in \{1, \ldots, p_2\}, j \in \{1, \ldots, p_1\} \\
\mathbf{W}_{ij}^{(3)} &\leftarrow \mathbf{W}_{ij}^{(3)} - \eta \boldsymbol{\delta}_j^{(3)} \mathbf{x}_i \quad \text{for all } i \in \{1, \ldots, d\}, j \in \{1, \ldots, p_2\}
\end{aligned}$$

There are a few useful points to highlight. First, notice that by starting at the output layer of the network, we can reuse computation. The variable $\boldsymbol{\delta}^{(2)}$ can use $\boldsymbol{\delta}^{(1)}$ and $\boldsymbol{\delta}^{(3)}$ can use $\boldsymbol{\delta}^{(2)}$, and so on if the network was deeper. Backpropagation is therefore gradient descent with a careful ordering of the gradient computation, to avoid inadvertently (and wastefully) recomputing terms. Second, we computed $\boldsymbol{\delta}_j^{(2)}$ before updating $\mathbf{W}_{ij}^{(1)}$. This was to ensure that we used the weights that produced the output in our update. If we updated $\mathbf{W}_{ij}^{(1)}$ first, then the gradient calculation for $\boldsymbol{\delta}^{(2)}$ would be incorrect.

You can see the pattern even just with a three layer neural network. In practice, we typically use packages that actually do automatic differentiation for us, instead of having to implement these particular formulas. But, it is important to understand the update rules underlying our algorithms so that we use them appropriately and to be able to hypothesize and debug why they might not be working.

9.3 Autoencoders and the Connection to PCA

At first glance, the latent factor approaches like PCA and neural networks seem quite unrelated. But there is in fact a connection. Both are trying to identify important factors or features. We can specify a neural network that actually has the same solution as PCA. This neural network is called an *autoencoder*, specifically one with a linear activation and a bottleneck layer—a lower dimensional hidden layer. We show this connection in this section, and then discuss the importance of this connection for understanding how neural networks learn data representations.

An autoencoder is a neural network where the targets are set to the inputs: for a given $\mathbf{x}$, we set $\mathbf{y} = \mathbf{x}$. For example, we might learn the NN using the loss $\|f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)} - \mathbf{x}\|_2^2$. The NN architecture is chosen so that information is discarded from $\mathbf{x}$, and the resulting output $\hat{\mathbf{y}}$ returns only the most significant signal in $\mathbf{x}$. One reason this is useful is to remove any noise that is in $\mathbf{x}$, so as not to fit to that noise. Another reason is that sometimes these targets are added as an additional auxiliary loss—in addition to a primary prediction—to encourage the network to learn features that both make accurate predictions and retain as much information in $\mathbf{x}$ as possible. This inductive bias reflects the hypothesis that such a representation should be better for generalization.

Now let us consider a linear autoencoder with one hidden layer. Such an NN has weights $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}$ and both transfers are the identity, resulting in the loss per-sample

$$\|\mathbf{x}_i\mathbf{W}^{(2)}\mathbf{W}^{(1)} - \mathbf{x}_i\|_2^2$$

The input dimension is d, the hidden dimension is p and the output dimension is d, giving $\mathbf{W}^{(2)} \in \mathbb{R}^{d\times p}$ and $\mathbf{W}^{(1)} \in \mathbb{R}^{p\times d}$. Notice that if $p \geq d$, then we can simply set $\mathbf{W}^{(2)}\mathbf{W}^{(1)} = \mathbf{I}$, and we trivially get zero error. We have to restrict the hidden layer to be a *bottleneck*, with $p < d$, to force the transformation to be lossy.[3] The resulting network learns to maintain only the most significant content in the inputs. To see why, notice that the optimization, for all the data, is

$$\min_{\mathbf{W}^{(2)}\in\mathbb{R}^{d\times p},\mathbf{W}^{(1)}\in\mathbb{R}^{p\times d}} \|\mathbf{X}\mathbf{W}^{(2)}\mathbf{W}^{(1)} - \mathbf{X}\|_2^2$$

[3]This objective is also called Reduced Rank Regression, when the targets more generally are any m targets where $p < d$ and $p < m$. The matrix $\mathbf{W} = \mathbf{W}^{(2)}\mathbf{W}^{(1)} \in \mathbb{R}^{d\times m}$ is rank p, namely is a reduced rank matrix, with the goal to learn such a reduced rank $\mathbf{W}$ to minimize $\|\mathbf{X}\mathbf{W} - \mathbf{Y}\|_2^2$.

Assume we have the SVD $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ with singular values $\sigma_1, \ldots, \sigma_d \geq 0$ and further assume that $\sigma_p > 0$ (namely at least the first p singular values are non-zero). Then we can select $\mathbf{W}^{(2)} = \mathbf{V}_p$ and $\mathbf{W}^{(1)} = \mathbf{V}_p^\top$ where $\mathbf{V}_p = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_p] \in \mathbb{R}^{d \times p}$ These weight matrices produce prediction $\hat{\mathbf{X}}$ where

$$\begin{aligned}
\hat{\mathbf{X}} &= \mathbf{X}\mathbf{W}^{(2)}\mathbf{W}^{(1)} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top\mathbf{V}_p\mathbf{V}_p^\top && \triangleright\ \mathbf{V}^\top\mathbf{V}_p = [\mathbf{I}_p; \mathbf{0}] \in \mathbb{R}^{d \times p} \\
&= \mathbf{U}[\mathbf{\Sigma}_p; \mathbf{0}]\mathbf{V}_p^\top && \triangleright\ \mathbf{\Sigma}[\mathbf{I}_p; \mathbf{0}] = [\mathbf{\Sigma}_p; \mathbf{0}] \in \mathbb{R}^{d \times p} \\
&= \mathbf{U}_p\mathbf{\Sigma}_p\mathbf{V}_p^\top
\end{aligned}$$

This solution is the best rank p approximation to $\mathbf{X}$, since we took the top p singular values and vectors, just as we did in PCA. Therefore, since these weights matrices give us the closest approximation we can get to $\mathbf{X}$ if we are restricted to at most rank p, we know that they are a solution to this optimization. This solution is the same as that returned by PCA: the top p right singular vectors of $\mathbf{X}$.

We can map this more directly to the matrix factorization view of PCA, where we discussed the dictionary $\mathbf{D}$ and representation $\mathbf{h}$. There we had $\mathbf{D} = \mathbf{V}_p^\top$ and $\mathbf{H} = \mathbf{U}_p\mathbf{\Sigma}_p$ composed of representations $\mathbf{h}_i$ as the rows. For this autoencoder, the new representation is the hidden layer, namely $\mathbf{h} = \mathbf{x}\mathbf{W}^{(2)}$. For all datapoints, this is $\mathbf{H} = \mathbf{X}\mathbf{W}^{(2)} = \mathbf{U}[\mathbf{\Sigma}_p; \mathbf{0}] = \mathbf{U}_p\mathbf{\Sigma}_p$ for the representation given by the autoencoder, which is the same as PCA.

As in PCA, this solution is not unique without further constraints. We can shift weight between $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ for the autoencoder, just as we can shift weight between $\mathbf{D}$ and $\mathbf{H}$. For PCA, if we opt for unit length dictionary elements—and so the features in $\mathbf{H}$ indicate the magnitude each component is used—we get the unique solution above. For autoencoders, if we enforce the constraint that the norm of $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are equal, then we get the solution above (up to a rotation matrix that does not change magnitudes).

Advanced Remark: Once we move beyond this linear setting and PCA, autoencoders and matrix factorization approaches are different. For example, sparse coding as a way to obtain a new representation $\mathbf{H}$ cannot obviously be written as a standard autoencoder with the typical activations where we use a linear weighting followed by a nonlinearity like sigmoid or ReLU. The reason for this is that these activations are quite restricted, and so cannot represent the sparse coding operation. In fact, if we consider more general operations in each layer, we can actually (somewhat trivially) recover sparse coding. Assuming we are still learning parameters $\mathbf{W}^{(2)}$ to produce the first hidden layer, we can define a new transformation to get layer one as $g(\mathbf{x}) = \text{argmin}_{\mathbf{h} \in \mathbb{R}^p} \|\mathbf{x} - \mathbf{h}\mathbf{W}^{(2)}\|_2^2 + \|\mathbf{h}\|_1$. Recall that this objective with an ℓ_1 regularizer on $\mathbf{h}$ is what is used by sparse coding to get a sparse representation $\mathbf{h}$.

The question becomes (a) are such layers useful and (b) how do we optimize for $\mathbf{W}^{(2)}$ through this optimization over $\mathbf{h}$? For the first question, one way they are useful is for the same reason that sparse coding is useful: they give more direct control on the properties of the representation. Further, this transformation is a nonlinear transformation that potentially adds more flexibility than the standard NN approach of linear operations followed by activations. For the second question, it is more complicated now to optimize through such a weird activation. It is no longer differentiable, and that was one of our criteria to make an activation easy-to-use. There are actually some nice ideas out there precisely on how to optimize through this weird activation, but this is much beyond the scope of this course.

Chapter 10

Generalization Error in More Settings

In this chapter, we dive deeper into questions around evaluating the generalization error of learning algorithms. We begin first with an overview of generalization error, and how this relates to our goal when learning models in practice. We then discuss the role that the optimizer can play in generalization error. Finally, we discuss the notion of generalization when we move beyond the simpler i.i.d. setting. The focus is on understanding the issues, and is light on solutions.

10.1 Bias, Variance and Generalization Error

Let us revisit bias and variance, and the connection to generalization error. When we talked about the bias and variance for linear regression, we assumed that the true model was linear, and so the only bias introduced was from the regularization. In reality, when using linear regression with regularization, we are introducing bias both from selecting a simpler function class and from the regularization. If the true function is not linear, then we cannot compare the learned weights for a linear function directly to the true function.

If a powerful basis is used to first transform the data, then we can learn nonlinear functions even though the solution uses linear regression. In this case, it is feasible that this function class is sufficiently powerful and includes the true function, and that the bias is mostly due to regularization. But, in general, it will be difficult to guarantee that we have specified a function class that includes the true function, and it will be difficult to directly compare our parameters to true parameters (which may not even be of the same dimension).

We can more generally talk about bias and variance by considering instead the reducible error. In fact, the bias-variance trade-off is all about reducing the reducible error. Recall that the generalization error for the squared error decomposes into the reducible and irreducible errors:

$$\mathrm{GE}(f) = \mathbb{E}[(f(X) - Y)^2] = \underbrace{\mathbb{E}[(f(X) - f^*(X))^2]}_{\text{reducible error}} + \underbrace{\mathbb{E}[(f^*(X) - Y)^2]}_{\text{irreducible error}} \tag{10.1}$$

where $f^*(x) = \mathbb{E}[Y|X = x]$. This f^* could be a highly nonlinear function, and may not be in our function class. For example, if we are learning a neural network with three hidden layers, each of size 1024, with ReLU activations, then f^* may not be in this set of functions.

We can write this reducible error in terms of the bias and variance of our learned function. We write $f_{\mathcal{D}}$ to emphasize that it is a random variable that depends on the dataset. We can first consider the bias for a given input $\mathbf{x}$,

$$\mathbb{E}\left[(f_{\mathcal{D}}(\mathbf{x}) - f^*(\mathbf{x}))^2\right] = \left(\mathbb{E}\left[f_{\mathcal{D}}(\mathbf{x})\right] - f^*(\mathbf{x})\right)^2 + \mathrm{Var}\left[f_{\mathcal{D}}(\mathbf{x})\right].$$

Notice that in the second line, the expectation is now inside the squared distance; this term corresponds to the squared bias. The bias here reflects the output of the estimated function $f_{\mathcal{D}}(\mathbf{x})$, in expectation across all datasets $\mathcal{D}$. The variance term reflects how much the prediction for $\mathbf{x}$ can vary, if we learn on different iid datasets. This decomposition of the mean-squared error into a squared bias and variance is not obvious, but does follow similar steps to above. It is left as an exercise. We can then write this more generally, in expectation over X as well

$$\mathbb{E}[(f_{\mathcal{D}}(\boldsymbol{X}) - f^*(\boldsymbol{X}))^2] = \mathbb{E}_{\boldsymbol{X}}\left[(\mathbb{E}_{\mathcal{D}}[f_{\mathcal{D}}(\boldsymbol{X})] - f^*(\boldsymbol{X}))^2 + \text{Var}_{\mathcal{D}}[f_{\mathcal{D}}(\boldsymbol{X})]\right] \tag{10.2}$$

where we subscript each expectation with the variable we are taking the expectation over, to be clear about the two sources of stochasticity.

A complex function class is likely to have low bias, but may have high variance because it can overfit to each dataset. This means across different datasets, we are likely to see very different functions and so the variance in the predictions $f(\mathbf{x})$ will also vary significantly. For example, if we have 100 data points for $d = 3$ dimensional inputs, and use a neural network with one million parameters, then likely we will have high variance. The bias is also likely low, since the true function for a three-dimensional input can likely be represented by such a complex neural network—though of course it is possible that it cannot and we still have some bias.

On the flip side, if the function class if very simple, the bias may be high and variance will be lower. Even though we saw that linear functions—or cubic functions which are still very simple—can overfit, this was only in the extreme case with very small samples sizes. With a reasonable number of samples, these functions will likely be relatively consistent. As we start expanding the complexity by using new features, like higher-order polynomials or kernels, then we start to get into much more complex function classes and may have high variance.

But these are just rules of thumb. We can reason about a few specific cases where we expect good performance. Let $\mathcal{F}$ be the class of function, f_{true} be the true function, n the number of samples, i.i.d. sampling and a reasonable optimization procedure to find $f \in \mathcal{F}$. Then we expect to have low variance and low bias in the following cases.

1. $\mathcal{F}$ is small (simple) and f_{true} is simple such that there is an $f \in \mathcal{F}$ that is similar to f_{true}.
2. $\mathcal{F}$ is big (complex) and n is very big.

The choice of $\mathcal{F}$ is exactly an inductive bias. Our goal is to constrain the function space so that we can best reduce the reducible error, namely identify the best approximation to the true function under data limitations.

Remark: The bias-variance analysis above considers the estimator in expectation across datasets. However, we may want stronger results. For example, we may want to know with high-probability that our learned function has certain properties. In other words, we want to know that the lower percentile over all these functions is still guaranteed to behave reasonably, not just the mean. Most generalization bounds are focused on exactly this. If you are interested in learning more, see Appendix A.9.

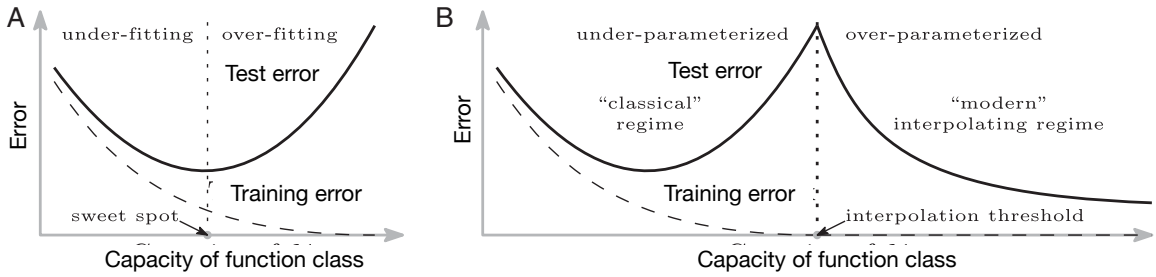


*Figure 10.1: The **double descent phenomena**. This picture is taken from [6] and lightly edited to match our notation. The x-axis is the capacity of our function class (hypothesis space) $\mathcal{F}$, where increasing to the right means we have a growing function class and can represent more functions. When we talk about underfitting and overfitting, we imagine the image on the left. If we underfit we have both higher train and test error. When we overfit, we have low train error and high test. In-between is a sweet spot where we have appropriately fit (learned from) our training set to generalize reasonably well, still typically with lower training error than test.*
The image on the right shows a new phenomenon where we consider the underparameterized regime (number of weights less than the number of samples) and the overparameterized regime (number of weights larger than the number of samples). We see the classic behavior in the underparameterized regime, but once we move to overparameterized regime, increasing capacity actually starts to reduce test error once again. See also [16] for a nice historical overview of this phenomenon.

10.2 Implicit Regularization with SGD & Large NNs

In Section 4.3, we saw that SGD can converge as quickly as batch GD with much less computation, as long as we are willing to be stuck with some error around the stationary point we reach. We discussed decaying the stepsize to get rid of this error. In practice, though, it is often effective to use optimizers like Adam or RMSProp that do not decay stepsizes to zero. So why isn't this error affecting us?

Recent evidence suggests that SGD acts like an implicit regularizer [26]. Empirically, it was found that SGD with larger learning rates improved generalization performance, even at the cost of increasing training error slightly. This phenomenon was explained by showing that SGD is actually optimizing a slightly different objective that includes a regularizer on the norm of the mini-batch gradients. The strength of this regularizer is controlled by the stepsize, and so a larger stepsize resulted in more implicit regularization.

Relatedly, this implies SGD prefers local minima in flatter rather than steep bowls. The preference for flatter bowls makes sense: the noise in SGD makes it hard to stay in a very steep bowl. Instead, it is likely to jump out of that bowl even with small perturbations to the weights. A flat bowl is harder to jump out of, with just noise. Note that implicit regularization could even improve performance for convex problems, but likely has a bigger impact for nonconvex objectives where we have these different local minima.

Further, using very large neural networks actually seems to improve generalization behavior. This phenomenon has been described as *double descent* [6], shown in Figure 10.1. This occurs in the *overparameterized* regime, where the number of weights is larger than

the number of samples. Such a scenario may seem outrageous, but modern networks are outrageously large. When we have smaller neural networks, then we can still underfit if it is very small and start to overfit to noise when it gets larger (but still underparameterized). As we start really increasing the number of parameters, we enter the overparameterized regime. Here, the test error starts to drop as we further increase the number of parameters.

All these networks have zero training error—they can perfectly fit the training dataset—so why does adding more parameters help? The answer is that the implicit regularization of SGD can identify better solutions amongst a broader set of candidates. In the overparameterized regime, we have many solutions that get zero training error, including the one that overfit right at switch from underparameterized to overparameterized. But with more parameters we are not stuck with this overfit solution, we have the flexibility to pick a better solution amongst the set of solutions. Intuitively, we want to pick the simplest solution amongst this set, since that one is least likely to overfit and most likely to match the true function and generalize well.

It is easier to understand this for fixed representations, where somewhat surprisingly this phenomenon also occurs. Imagine we use RBF features, where we randomly sample centers. We can increase the dimensionality by simply sampling more centers p, until $p > n$, and so also get into the overparameterized regime. With linear regression, we have more degrees of freedom than needed and we can directly solve for $\mathbf{\Phi w} = \mathbf{y}$ for $\mathbf{\Phi} \in \mathbb{R}^{n \times p}$. In fact, there are infinitely many solutions $\mathbf{w}$ that exactly get $\mathbf{\Phi w} = \mathbf{y}$. We know that larger weights can indicate overfitting—recall our SVD analysis—and so amongst these we could select the solution with the smallest ℓ_2-norm on the weights. This solution is likely to generalize better. This has actually been shown empirically (see [6]). In the underparameterized regime ($p < n$), as overfitting starts to occur with growing p, we also see the norm of the weights increasing. Once we reach the overparameterized regime ($p > n$), the norm actually starts to decrease and we see test error start to improve. The overall conclusion is that we can get better solution quality with a larger hypothesis space and (implicit) regularization.

An additional benefit beyond better solution quality is that optimization is easier with very large neural networks. This result may again be counterintuitive at first, but actually makes a lot of sense. Learning in a higher-dimensional space can make the optimization easier because there are more dimensions to find error reduction. SGD is a local search, and it needs to find a way to locally change its parameters to further reduce the loss. With more dimensions, it is more likely that there is a path to further error reduction from it's current point. Additionally, in a very high-dimensional space, it is more likely for a random initialization to start near a good solution, making it easier for SGD to get to that nearby good solution. These optimization benefits help explain why it can be much simpler to train very large models, and then prune or compress them afterwards (see [11] for an in-depth discussion).

Once again, behavior in higher-dimensions defies our expectations and these outcomes are an instance of the blessing of dimensionality. Learning in a higher-dimensional space makes the optimization simpler—the local search by SGD is effective. The combination of the SGD optimizer and overparameterized models, like large neural networks, finds simpler solutions amongst the space of solutions that generalize well. There is much more work here to understand these unexpected phenomena, but SGD with non-decaying stepsizes on large neural networks currently seems like a promising approach to obtain fast convergence and good generalization.

10.3 Moving Beyond the iid Setting

We have so far assumed that we obtain an i.i.d. sample from the underlying distribution. In practice, however, we know that there will be some violation of this assumption. We can step back and ask what we really mean by generalization in this practical setting, rather than what is convenient to analyze. Two properties that characterize data in the real-world are that distributions change (slowly) over time (*nonstationarity*) and our data collection is biased towards a particular subset of the data (*covariate shift*). These two issues are related, but slightly different.

10.3.1 Generalization Issues under Covariate Shift

Let us open up this section with an example. Imagine you gather images of a room from April to October, in Edmonton. A lot of the data will have quite a bit of light from the long days, but you will still get to see many images in both dark and light conditions. Then you test the predictions on images from January to March. The images will generally be darker, and so the model trained on many images where the room has more light might not be as effective. Because many of the images in the training data were taken under conditions with more light, the predictor implicitly put more weight on getting such images correct, potentially to the detriment of the accuracy for the images under lower light conditions.

This setting is typically called *covariate shift*, and it is reasonably feasible to address with reweighting schemes. The term covariate shift implies only that there is a shift in the distribution over $\mathbf{x}$, but $p(y|\mathbf{x})$ stays the same. This is exactly what occurs in our image example. Predicting if an image contains a person or not, conditioned on the image, remains the same regardless of it is April or January. In other words, $p(y = \text{Has Person}|\mathbf{x} = \text{Room Snapshot})$ does not change. But, the distribution over the room snapshots that we see does change, namely over $p(\mathbf{x} = \text{Room Snapshot})$.

To see why this can be corrected with reweighting, let the test distribution over images be p_{test} and p_{train} for the training data. The GE for this setting is

$$\text{GE}(f) = \mathbb{E}_{p_{\text{test}}}[(f(X) - Y)^2] = \int_{\mathcal{X}} p_{\text{test}}(\mathbf{x})\mathbb{E}[(f(\mathbf{x}) - Y)^2|X = \mathbf{x}]d\mathbf{x} \tag{10.3}$$

In other words, the generalization error is the error across all pairs under distribution $p(y|\mathbf{x})p_{\text{test}}(\mathbf{x})$. When we minimize the squared error on training data obtained using p_{train}, we are instead trying to minimize the error across all pairs under distribution $p(y|\mathbf{x})p_{\text{train}}(\mathbf{x})$.

$$\int_{\mathcal{X}} p_{\text{train}}(\mathbf{x})\mathbb{E}[(f(\mathbf{x}) - Y)^2|X = \mathbf{x}]d\mathbf{x}$$

We simply need to reweight the importance of a sample $(\mathbf{x}, y)$ using[1] $p_{\text{test}}(\mathbf{x})/p_{\text{train}}(\mathbf{x})$. We saw how to incorporate weightings into regression, in Section 3.1.1.

Exercise 30: Consider the weighted squared error loss

$$c(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} b_i \left(f_{\mathbf{w}}(\mathbf{x}_i) - y_i\right)^2,$$

[1] You may notice an additional complication here that we may not have access to either of these distributions, p_{test} nor p_{train}. A large part of the literature on covariate shift is about estimating these reweightings, without having these distributions explicitly. The goal here was to introduce you to the problem setting, and so we do not cover these approaches here.

where we use $b_i = p_{\text{test}}(\mathbf{x}_i)/p_{\text{train}}(\mathbf{x}_i)$. Show that in expectation, across (X, Y) sampled according to $p(\mathbf{x}, y) = p(y|\mathbf{x})p_{\text{train}}(\mathbf{x})$, that this loss equals the true generalization error in Equation (10.3). □

10.3.2 Issues of Data Coverage and Using Inductive Biases

More difficult is the setting when the distribution of the training data does not cover what is observed in deployment. For this setting, we have $\mathbf{x}$ where $p_{\text{test}}(\mathbf{x}) > 0$ but $p_{\text{train}}(\mathbf{x}) = 0$. For example, let us imagine a setting where images were only collected during the day, when taking Room Snapshots. But, now, we would like to recognize if there is a person in the room under very low-light, in the evening. The training data does not contain any images in the evening, and so we might wonder if such a generalization task is even possible.

The answer to this question depends heavily on what we build into our model. What we build is in typically called an *inductive bias*. A prior is an inductive bias, as is the optimization algorithm we use to find our parameters, as is the architecture we use for our neural network. It is anything that defines the learning algorithm, before we feed data into it. In the above example, we could design an architecture that focuses on edges in an image, and attempts to remove information in the pixels that is due to different lighting conditions. Under such an architecture, it is feasible that we could learn a model on data that only has images in the daytime, and deploy on test data that includes images in the evening. Our inductive bias makes it so that training datapoints are now representative of testing datapoints. It allows us to say that two images are similar—even if their lighting conditions are different—which means we can generalize our predictions between these two images.

The way our model generalizes, therefore, depends on both the inductive biases and the available data. If we want our model to generalize well, then we have to consider both. If we know we will have lots and lots of data, covering many different scenarios, then we as designers do not need to build in as much. More can be learned from data, and a good hypothesis identified from a large class of hypotheses. If we know we will have limited data, then we know that likely we need to build in more. The data does not allow us to sufficiently narrow the class of hypotheses. Ideally, we build in just enough to allow the learner to generalize faster, with a minimal amount of data (be sample efficient). But, we do not want to build in too many biases, since they may be incorrect and so limit the ability to learn a very good model as we get more and more data.

In summary, for all three of these settings—i.i.d. data, covariate shift with coverage and covariate shift without coverage—we have a similar approach. We learn on training data, with the goal to perform well on new unseen data. The difficulty of generalizing to this unseen data is simply harder when moving from i.i.d. data, to covariate shift and to settings where we only see a restricted subset of possible inputs (lack of coverage). In all three settings, the unseen data is not the same as the training data, and inductive biases allow us to overcome this gap. The bigger the mismatch between training and test, the more we have to consider what assumptions to build in to facilitate generalization. But, in all three cases, it is key to consider the inductive biases, since they are crucial to generalization.

Remark: The other terms typically used for understanding generalization are interpolation and extrapolation. Intuitively, interpolation means that we make predictions between datapoints, and extrapolation is outside our datapoints. Such definitions require a notion

of what is considered to be between our datapoints and what is considered to be outside our datapoints. Instead, it is more direct to reason about inductive biases and how they relate to the available data. If we really want to map to these terms, we can say that our learning algorithms are interpolating across the provided data and our inductive biases allow us to extrapolate beyond the data.

10.3.3 Nonstationarity and Generalization

The relationship between $\mathbf{x}$ and y may itself be nonstationary. This means that $p(y|\mathbf{x})$ may change between training and test—$p_{\text{train}}(y|\mathbf{x}) \neq p_{\text{test}}(y|\mathbf{x})$. Consider the stock market. We may have data from the last 20 years. However, the world and economy is constantly changing, and this data is not perfectly predictive of what will occur in the next year. For one, we always have an increasing trend in the total value of stocks. But, more importantly, sudden technological or societal changes can have unexpected consequences that are simply not in the data, because they have never been observed.

Many of our datasets have some level of nonstationarity, because they are actually measured across time. The level of nonstationarity can be small. For example, we can consider predictions for the required pump speed in a home heating system, conditioned on readings in the current system (e.g., desired temperature, temperature outside, etc.). This system is relatively self-contained, with consistent predictions conditioned on all of the readings in the system. However, slowly over time a pipe might start to get dirty, making it necessary for the pump speed to slowly increase to account for this change. This change occurs very slowly over a long window of time, but nonetheless is not visible to the model and so the data appears nonstationary. If we consider y to be the pump speed and $\mathbf{x}$ the readings in the system, then this means that $p(y|\mathbf{x})$ is slowly changing over time.

We can overcome some of this nonstationarity by attempting to model the trend in that nonstationarity. For example, in addition to $\mathbf{x}$, we can input the average pump speed over a recent history. If it has recently increased, the prediction for the pump speed for this current $\mathbf{x}$ could also be increased in the prediction. Augmenting with history can help overcome the partial observability in the system, that makes it appear nonstationary; we will discuss this more in Chapter 16.

However, we will not be able to overcome all nonstationarity in this way, for the same reasons as above: a lack of data coverage. For example, when training the system to predict pump speeds, we may never have observed the system under any degradation, such as having accumulated dirt in the pipes. We cannot learn from the training data that a short history of pump speeds allows us to account for this change. Instead, it is key to allow the model to continue to update with new data. This approach is called *tracking*.

This last setting leads us to one other important distinction in generalization: *static* and *dynamic* generalization. (This distinction is sometimes called zero-shot and few-shot generalization). In the static setting, we want our function—say our learned neural network—to directly generalize to the test deployment setting. In the dynamic setting, we want our learned data representation to facilitate further learning. In other words, we want it to allow the function to update with as few samples as possible.

Example 15: [The role of representations in sample efficiency] Let us consider a simple example where the features can help us learn more efficiently. This example emphasizes the utility in having a compact set of features that are used for many inputs, to promote

generalization. Assume our data is actually generated by a linear function $Y = xw + \epsilon$, for a scalar input x and $\epsilon \sim \mathcal{N}(0, 1)$. But, we didn't know the true function would be so simple. We created features that we hoped would let us fit more complex nonlinear functions: a fine-grained binning. The features are $\phi_j(x) = 1$ is x is in bin j, and zero otherwise. Assuming $x \in [-1, 1]$, and we use 200 bins, then we have that $\phi_1(x) = 1$ if $x \in [-1, -0.99)$, $\phi_2(x) = 1$ if $x \in [0.99, -0.98)$, and so on. This feature vector $\boldsymbol{\phi}(x) \in \mathbb{R}^{200}$ is an indicator vector for which bin x is in.

These features allow us to learn a different $\mathbb{E}[Y|x]$ for each bin, which is a highly nonlinear—and even discontinuous—function. However, these features make learning very slow. If we had used the original x as features, then we would have learned very quickly. Likely, we would find w with very few samples; even after 10 samples, we'd likely have a reasonable estimate. For these binning features, however, we need at least one sample per bin to even obtain an estimate of $\mathbb{E}[Y|x]$ for x in that range. That means we need at least 200 samples, but likely more. □

Understanding how data representations impact the update, and generally how we can learn representations that tackle some of these generalization goals, is still in its infancy. The goal here is just for you to understand and appreciate the different generalization goals.

Part III

Generative Models

Up until now, we have been focused on *predictive models*, ones that learn $p(y|\mathbf{x})$ for a relatively simple conditional distribution (exponential family). In this section, we discuss *generative models*, that attempt to learn more complex distributions.

We have already seen very simple generative models, when first talking about MLE. Recall that our goal was to model distribution $p(\mathbf{x})$ with parameters $\boldsymbol{\theta}$, assuming we were given a dataset $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n$. In MLE, the goal is to find the parameters that make the data the most likely

$$\textbf{MLE} : \theta_{\text{mle}} = \operatorname*{argmax}_{\boldsymbol{\theta}} p(\mathcal{D}|\boldsymbol{\theta}) = \operatorname*{argmin}_{\boldsymbol{\theta}} \sum_{i=1}^{n} -\ln p(\mathbf{x}_i|\boldsymbol{\theta}) \tag{10.4}$$

where we previously saw that this could be rewritten as the minimization of the negative log likelihood. For the distributions we considered, like Gaussians and Poissons, the log simplified the objective substantially and the resulting gradients were easy to compute. So, we did not dwell too much on learning algorithms for these models.

Once we move to more complex distributions, however, learning is more complicated. Further, we have an additional criterion: the distributions need to be efficient to sample from. For our simple distributions, like Gaussians, it is straightforward to get samples of $\mathbf{x}$, so again we did not dwell on it. For generative models, our primary goal will be to sample $\mathbf{x} \sim p_{\boldsymbol{\theta}}$. Generative models are designed around these two criteria: facilitating sampling and facilitating learning the model.

In this part, we start with one of the simplest generative models that already adds quite a bit of complexity: mixture models. Then we talk about how to incorporate one of the central tools in these notes: data representations.

Chapter 11

Simple Generative Models: Mixture Models

In this chapter we discuss how to use and learn mixture models. Let's assume that each mixture component has parameters $\boldsymbol{\theta}_k$, giving

$$p(\mathbf{x}|\boldsymbol{\theta}) = \sum_{k=1}^{m} w_k p(\mathbf{x}|\boldsymbol{\theta}_k) \quad \text{where} \quad \boldsymbol{\theta} = (w_1, w_2, \ldots, w_m, \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_m).$$

For example, each component might be a Gaussian distribution with parameters $\boldsymbol{\theta}_k = (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$. The mixture distribution corresponds to a convex combination of these Gaussians with different means and covariances.

We already saw in Section 2.3 how this class of distributions can significantly increase modeling power. They provide enough complexity to be used as generative models, to model realistic distributions and make it useful to simulate/generate potential outcomes using these learned distributions. We first discuss how mixture models are used as generative models and then discuss how to learn them.

11.1 Using Mixture Models

For a generative model to be useful, we need to be able to efficiently obtain samples from it. Using a generative model means getting a sample from a generative model. (Using a prediction model meant getting a single, likely outcome given context). One advantage of mixture models is that they are very easy to sample from, as long as it is easy to sample from each component $x \sim p_k$.

The sampling procedure is summarized in Algorithm 5. First we sample k proportionally to probabilities $p[k] = w_k$, shown in Algorithm 6. The idea for this algorithm is simple. We discretize the interval $[0, 1]$ into m buckets, with the first bucket of size w_1, the second of size w_2 and so on. Then we uniformly randomly pick a number in the range $[0, 1]$, and return the bucket that u falls into. In expectation, we will see $k = 1$ picked w_1 percentage of the time, and $k = 2$ picked w_2 percentage of the time, and $k = 3$ picked w_3 percentage of the time, and so on.

Once we have a component k selected, then we sample $x \sim p_k$. This second step is also simple, as long as each p_k is easy to sample from. This is the case for many of the simple distributions we considered. A common procedure, when the inverse CDF exists, is to sample $u \sim [0, 1]$ uniformly and then transform it using the inverse CDF, as shown in Algorithm 7. Recall that the CDF F for a random variable X is defined as $F(x) = \Pr(X \leq x)$. Notice that $U = F(X)$ is a uniform random variable on $[0, 1]$. Because the CDF is invertible, we have that $X = F^{-1}(U)$. This means we can sample from a uniform U, and apply the transformation F^{-1}, so it is as if we sampled directly from X. In reality, you

Algorithm 5: Sample from Mixture Model

Input: $\mathbf{w} \in [0,1]^m$ and mixture components $p_1, \ldots, p_m$
$k \leftarrow$ Sample from Categorial Distribution($\mathbf{w}$) (see Algorithm 6)
$\mathbf{x} \leftarrow$ Sample from Component Distribution p_k with inverse CDF F_k^{-1} (see Alg 7)
return $\mathbf{x}$

Algorithm 6: Sample from Categorical Distribution

Input: categorical probabilities $\mathbf{w} \in [0,1]^m$, such that $\sum_{k=1}^m w_k = 1$
Sample u uniformly from $[0,1]$
Set $s = 0$
for $k = 1$ to $m-1$ **do**
    $s \leftarrow s + w_k$
    if $s \geq u$, return k
return m

would not implement Algorithm 7, as such standard sampling approaches are available in most code packages/libraries. For example, in Python in numpy, to sample from a Gaussian, you would use `numpy.random.normal`.

Exercise 31: Show that Algorithm 6 always returns a valid $k \in \{1, \ldots, m\}$. □

Exercise 32: For a random variable X that has CDF F, show that $U = F(X)$ is a uniform random variable on $[0,1]$ □

Example 16: Let us consider one use case for using mixture models as generative models: synthetic data for health care. To avoid sharing patient data, we can instead learn a generative model on that data and only share synthetic samples from that generative model.[1] Imagine the data consists of $d = 10$ attributes, including height, weight, and health metrics like cholesterol levels and hemoglobin levels, for $n = 1000$ patients. Also imagine someone has applied the learning procedure described in the next section, to get back a Gaussian mixture model with $m = 10$ Gaussian components $p_1, \ldots, p_m$ each with parameters $\boldsymbol{\theta}_1 = (\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1), \ldots, \boldsymbol{\theta}_m = (\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$.

We want to generate a new dataset of 100 plausible patients, to examine correlations between health metrics and height. To do so, we call Algorithm 5 one hundred times, and store the sampled $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{100}$. These synthetic vectors should represent plausible patients, and the correlations in the data used to create the mixture model. For example, if height is often associated with higher hemogloblin, then we should see a pattern that generated patients $\mathbf{x}_i$ that are taller also tend to have higher hemoglobin. It is unlikely that $\mathbf{x}_i$ will correspond to any actual patient, but rather will look like the patients in the underlying dataset. □

[1] Using synthetic data is actually one avenue currently being pursued to benefit from health data without sharing patient data, albeit with also smarter strategies to ensure the synthetic samples do not compromise privacy.

Algorithm 7: Sample using Inverse CDF

1: **Input**: inverse CDF F^{-1}
2: Sample u uniformly from $[0, 1]$
3: **return** $F^{-1}(u)$

11.2 Learning Mixture Models

Now let us turn to learning mixture models. This ends up being more complicated than our typical maximum likelihood procedure. First, let's try to simplify the log likelihood for a single point.

$$-\ln p(\mathbf{x}_i|\boldsymbol{\theta}) = -\ln \sum_{k=1}^{m} w_k p(\mathbf{x}|\boldsymbol{\theta}_k)$$

Immediately we can notice that the log is not going to help as much as before. Previously the log cancelled the exponential terms in the simpler distributions, like the single Gaussian, making the resulting objective simple. For example, for a Gaussian, our objective became a sum of squared errors. This may not seem like an issue: after all, we can still take the gradient of this objective and do gradient descent. In fact, such an algorithm has been used. But, the issue seems to be that this approach results in slow convergence.

Instead, we will take a different route and introduce auxiliary variables that correspond to the mixing component. We jointly optimize over these auxiliary variables and the original variables $\boldsymbol{\theta}$, to obtain simpler iterative updates that nonetheless still converges to a stationary point of the above objective. This algorithm—rederived in many forms—is called *expectation-maximization*. In this section, we will simply provide the algorithm, and relegate the derivation to Appendix A.7. The reason for this is that we can actually more simply obtain this algorithm once we see the more general ELBO loss that we also use for variational auto-encoders in Chapter 12. Here, the primary goal is to get exposed to this algorithm, before jumping into the hairy details.

The EM algorithm is based on iteratively applying the following two steps. The **E-step** corresponds to updating the probability array $p_t[i, k] =$ probability that sample i came from component k. Mathematically, this corresponds to

$$p_t[i, k] \stackrel{\text{def}}{=} \Pr(k|\mathbf{x}_i) = \frac{\Pr(\mathbf{x}_i|k)\Pr(\mathbf{x}_i)}{\Pr(k)} = \frac{w_k^{(t)} p(\mathbf{x}_i|\boldsymbol{\theta}_k^{(t)})}{\sum_{j=1}^{m} w_j^{(t)} p(\mathbf{x}_i|\boldsymbol{\theta}_j^{(t)})} \tag{11.1}$$

where the probabilities Pr use the current parameters for our mixture model, $\boldsymbol{\theta}^{(t)}$. In other words, $\Pr(\mathbf{x}_i|k)$ is actually $\Pr(\mathbf{x}_i|k, \boldsymbol{\theta}^{(t)})$, $\Pr(k)$ is actually $\Pr(k|\boldsymbol{\theta}^{(t)})$, etc. We leave the last step as an exercise.

Exercise 33: Show the steps to get the last equality in Equation 11.1. □

The **M-step** corresponds to updating the model parameters, by solving the following

Algorithm 8: EM for Gaussian Mixture Models

Input: number of components m
Initialize $\boldsymbol{\mu}_k^{(0)}$, $\boldsymbol{\Sigma}_k^{(0)}$ and $w_k^{(0)}$ for all $k \in 1$ to m, $t = 0$
while not converged **do**
  $p_t[i,k] = \frac{w_k^{(t)} p(\mathbf{x}_i|\boldsymbol{\theta}_k^{(t)})}{\sum_{j=1}^m w_j^{(t)} p(\mathbf{x}_i|\boldsymbol{\theta}_j^{(t)})}$ for all $i \in \{1,2,\ldots,n\}$, $k \in \{1,2,\ldots,m\}$
  Compute $p_t[k] \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^n p_t[i,k]$
  for $k \in \{1,2,\ldots,m\}$ **do**
    $w_k^{(t+1)} = p_t[k]$
    $\boldsymbol{\mu}_k^{(t+1)} = \frac{1}{np_t[k]}\sum_{i=1}^n p_t[i,k]\mathbf{x}_i$
    $\boldsymbol{\Sigma}_k^{(t+1)} = \frac{1}{np_t[k]}\sum_{i=1}^n p_t[i,k](\mathbf{x}-\boldsymbol{\mu}_k^{(t+1)})(\mathbf{x}-\boldsymbol{\mu}_k^{(t+1)})^\top$
  $t \leftarrow t+1$
return $w_k^t, \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)}$ for all $k \in \{1,2,\ldots,m\}$

minimization problems, where $\mathcal{F}$ is the simplex.

$$\operatorname*{argmin}_{\mathbf{w}\in\mathcal{F}} -\sum_{k=1}^m p_t[k] \ln w_k \tag{11.2}$$

$$\operatorname*{argmin}_{\boldsymbol{\theta}_k} -\sum_{i=1}^n p_t[i,k] \ln p(\mathbf{x}_i|\boldsymbol{\theta}_k) \tag{11.3}$$

where we slightly overload notation and define array $p_t[k] \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^n p_t[i,k]$. Notice that $p_t[k]$ is the average likelihood of each component across samples. The beauty of the EM formulation is that the optimization for each component distribution is independent, for the given probabilities $p_t[i,k]$ computed in the E-step. (Again, to understand why you need to look at the derivation in Appendix A.7 or wait to see the VAE derivation and Section 12.3). We are already pros at solving (weighted) log likelihood problems for simple distributions, so solving for each $\boldsymbol{\theta}_k$ is simple. We have also already seen how to solve for $\mathbf{w}$, with simplex constraints, in Section 6.3, where we found that intuitively $w_k = p_t[k]$, the average likelihood of each component across samples.

The above was all agnostic to the choice of component distributions. To be concrete, we summarize the EM algorithm for a mixture of m Gaussian distributions in Algorithm 9. A common convergence criteria is to check if $\sum_{k=1}^m p_t[k] \ln w_k$ barely changed between iterations. This criteria checks that the component weighting have converged, and is sufficient to indicate that the whole optimization has converged.

Exercise 34: Derive the updates to the Gaussian parameters in Algorithm 9. □

Exercise 35: A typical heuristic is to initialize the coefficients to be uniform, the covariance matrices to be diagonal with a large number on the diagonal and to initialize the means to random points in the dataset. What might happen if we initialized the covariance matrices to be very small instead? □

Algorithm 9 can be modified to use other component distributions, simply by solving Equation 8 for your chosen distribution. You can even have different distributions for

Algorithm 9: EM for any component distribution

Input: number of components m, with components distributions $p(\cdot|\boldsymbol{\theta}_1), \ldots, p(\cdot|\boldsymbol{\theta}_m)$
Initialize $\theta_k^{(0)}$ and $w_k^{(0)}$ for all $k \in 1$ to m, $t = 0$
while not converged **do**
  $p_t[i,k] = \frac{w_k^{(t)} p(\mathbf{x}_i|\boldsymbol{\theta}_k^{(t)})}{\sum_{j=1}^m w_j^{(t)} p(\mathbf{x}_i|\boldsymbol{\theta}_j^{(t)})}$ for all $i \in \{1,2,\ldots,n\}$, $k \in \{1,2,\ldots,m\}$
  Compute $p_t[k] \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^n p_t[i,k]$
  for $k \in \{1,2,\ldots,m\}$ **do**
    $w_k^{(t+1)} = p_t[k]$
    $\boldsymbol{\theta}_k^{(t+1)} = \text{argmin}_{\boldsymbol{\theta}_k} - \sum_{i=1}^n p_t[i,k] \ln p(\mathbf{x}_i|\boldsymbol{\theta}_k)$
  $t \leftarrow t+1$
return $w_k^t, \boldsymbol{\theta}_k^{(t)}$ for all $k \in \{1,2,\ldots,m\}$

different components, since they are solved independently. To be concrete, we provided the updates for Gaussian distributions, but the above can be generically done for other distribution. Similar update rules can be obtained for different probability distributions, where the derivatives for the mixture parameters will be slightly different but the solution for the coefficients $\mathbf{w}$ is actually the same.

Exercise 36: Rewrite Algorithm 9 with component distributions corresponding to exponential distributions. □

Exercise 37: Rewrite Algorithm 9 with two components distributions, one Gaussian and one exponential. □

Exercise 38: We could also use a mixture model with component distributions that are categorical. For example, for a discrete random variable with categories $\{1,2,\ldots,s\}$ with $s = 25$, we could define a mixture model with $m = 3$ and three categorical distributions. Show that this does not provide additional modeling power beyond using a single categorical distribution. □

Chapter 12

Generative Models using Data Representations

In this chapter we discuss how the data representation approaches allow us to learn more complex generative models, not just more complex predictive models. We have already seen how to learn generative models by making simple parametric assumptions on $\mathbf{x}$, such as assuming $\mathbf{x}$ is Gaussian or that it is a mixture model. Even with the generalization to mixture models, however, these models can be quite limited, either requiring a large number of mixtures, requiring a smarter distance than Euclidean distance or requiring careful tuning of the number of mixture components.

We first discuss one simple way to improve the capacity of these models: mapping to a new space with a data representation, and then using simpler parametric models. We then discuss how to directly learn complex distributions, with neural networks, using the idea of *reparameterization*.

12.1 Connections to Models We Have Already Discussed

We have two goals with generative models: learning a good approximation $\hat{p}(x)$ to a potentially complex distribution $p(x)$ and having an efficient approach to sample from $\hat{p}$. When selecting our approximation strategy, we have to keep both in mind. We have actually already seen several models where generating samples is straightforward, including mixture models and probabilistic PCA.

Let us revisit probabilistic PCA. Recall that we assumed a latent $\mathbf{h} \in \mathbb{R}^p$ where $p(\mathbf{x}|\mathbf{D}) = \int p(\mathbf{x}|\mathbf{h}, \mathbf{D})p(\mathbf{h})d\mathbf{h}$. In particular, we assumed that $p(\mathbf{x}|\mathbf{h}, \mathbf{D}) = \mathcal{N}(\boldsymbol{\mu} = \mathbf{h}\mathbf{D}, \sigma^2\mathbf{I})$ for some $\sigma > 0$ and that $p(\mathbf{h}) = \mathcal{N}(\boldsymbol{\mu} = \mathbf{0}, \mathbf{I})$. We discussed probabilistic PCA as a way to identify these latent factors, but it is also a way to obtain a generative model, namely $p(\mathbf{x}|\mathbf{D})$. Like mixture models, this model is easy to sample from, because it is broken up into sampling the latent component—here a continuous vector rather than a discrete index—and then sampling from the simpler distribution given this latent component—a Gaussian $p(\mathbf{x}|\mathbf{h}, \mathbf{D})$. The procedure is

1. For $k = 1, 2, \ldots, p$, sample $h_k \sim \mathcal{N}(0, 1)$
2. Let $\mathbf{h} = [h_1, h_2, \ldots, h_p]$
3. Sample $\mathbf{x} \sim \mathcal{N}(\mathbf{h}\mathbf{D}, \sigma^2\mathbf{I})$

This generative model, however, is quite limited in terms of the distributions it can represent over $\mathbf{x}$. This is particularly due to the simplistic assumption on $p(\mathbf{x}|\mathbf{h}, \mathbf{D})$. However, this is precisely the term we know how to generalize, using data representations like neural networks! In particular, instead of assuming the mean is linear in $\mathbf{h}$, we can assume it is

a more complex function $f(\mathbf{h})$. Effectively, $p(\mathbf{x}|\mathbf{h}, \mathbf{D})$ is a regression problem where $\mathbf{h}$ are the inputs, $\mathbf{D}$ are the parameters of the learned function and $\mathbf{x}$ are the outputs. This is the strategy taken by variational autoencoders, that we discuss next.

12.2 Variational Autoencoders

In this section, we see one of the simplest generative models that uses the power of neural networks, namely as a simple extension of probabilistic PCA. These models, called Variational Autoencoders (VAEs), obtain stochasticity with simple normal random variables and use complex mappings (neural networks) to transform those simple random variables into complex distributions. For example, consider an f that is a higher-order polynomial. Then $\mathbf{x} = f(\mathbf{h})$ map each normal $\mathbf{h}$ to a completely different vector and the resulting random variable is no longer Gaussian and can be a highly complex distribution.

For VAEs, we assume $p(\mathbf{x}|\mathbf{W}) = \int p(\mathbf{x}|\mathbf{h}, \mathbf{W})p(\mathbf{h})d\mathbf{h}$ with $p(\mathbf{h}) = \mathcal{N}(\boldsymbol{\mu} = \mathbf{0}, \mathbf{I})$ and $p(\mathbf{x}|\mathbf{h}, \mathbf{W}) = \mathcal{N}(\boldsymbol{\mu} = f_{\mathbf{W}}(\mathbf{h}), \sigma^2\mathbf{I})$ for some $\sigma > 0$, where $f_{\mathbf{W}}$ is a multi-layer neural network input $\mathbf{h}$ and outputting a d-dimensional vector, the same size as $\mathbf{x}$. The primary difficulty is in learning $\mathbf{W}$, but first let us consider how we use the VAE. The procedure is

1. For $k = 1, 2, \ldots, p$, sample $h_k \sim \mathcal{N}(0, 1)$
2. Let $\mathbf{h} = [h_1, h_2, \ldots, h_p]$
3. $\boldsymbol{\mu} = f_{\mathbf{W}}(\mathbf{h})$
4. Sample $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2\mathbf{I})$

It is almost exactly the same as the mode from probabilistic PCA, but the mean is a multi-layer neural network rather than a linear function of $\mathbf{h}$.

Our goal is to find $\mathbf{W}$ that maximizes the likelihood of an observed dataset $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^n$. As with mixture models, we will run into the same issue that the latent variable will make estimation more complex. For mixture models, the log was stuck around the outside of a sum; here, the log will be stuck around the outside of the integral over $\mathbf{h}$. As in EM, directly reasoning about the probabilities over these hidden variables conditioned on the inputs $\mathbf{x}$, will help us significantly simplify this optimization.

Similar to probabilistic PCA (PPCA), our density is

$$p(\mathbf{x}|\mathbf{W}) = \int p(\mathbf{x}|\mathbf{h}, \mathbf{W})p(\mathbf{h})d\mathbf{h} = \mathbb{E}_{\mathbf{h}\sim p}[p(\mathbf{x}|\mathbf{h}, \mathbf{W})]$$

where the integral takes the expectation over $\mathbf{h}$, where we explicitly subscript the expectation to be clear from which distribution we are sampling $\mathbf{h}$. The density is similar to PPCA, but now it does not evaluate to a Gaussian, since $p(\mathbf{x}|\mathbf{h}, \mathbf{W})$ is more complex. Our goal is to maximize the likelihood of the data (minimize the negative log-likelihood)

$$\operatorname*{argmin}_{\mathbf{W}} -\sum_{i=1}^{n} \ln p(\mathbf{x}_i|\mathbf{W})$$

We can attempt to estimate $p(\mathbf{x}_i|\mathbf{W})$ by sampling many $\mathbf{h}_1, \ldots, \mathbf{h}_m \sim p(\mathbf{h})$, and using $\frac{1}{m}\sum_{k=1}^m p(\mathbf{x}_i|\mathbf{h}_k, \mathbf{W}) \approx \mathbb{E}_{\mathbf{h}\sim p}[p(\mathbf{x}|\mathbf{h}, \mathbf{W})] = p(\mathbf{x}_i|\mathbf{W})$.

The issue with this approach is that many $\mathbf{h}$ play little to no role in the expectation. Namely, $p(\mathbf{x}_i|\mathbf{h}, \mathbf{W})$ is likely very near zero for most $\mathbf{h}$. Notice that the mean for $p(\mathbf{x}|\mathbf{h}, \mathbf{W})$ is $f_{\mathbf{W}}(\mathbf{h})$. This mean will be close to some $\mathbf{x}$, but most $\mathbf{x}$ will be far from $f_{\mathbf{W}}(\mathbf{h})$ and so the density at those points will be very small. We would have to sample many many $\mathbf{h}$ to get an accurate estimate of this expectation, because only a small number of those sampled $\mathbf{h}$ will be in the region where $p(\mathbf{x}_i|\mathbf{h}, \mathbf{W})$ is reasonably large.

Instead, we can try to direct the sampling, so that we sample exactly these $\mathbf{h}$ that are pertinent to $\mathbf{x}_i$. To do so, we need a distribution $q(\mathbf{h}|\mathbf{x})$, so that we can sample $\mathbf{h}$ with high likelihood for a given $\mathbf{x}$. We do not have such a distribution, but we can attempt to *learn* it. Note that this distribution $q(\mathbf{h}|\mathbf{x})$ is typically called the *variational* distribution, giving us part of the name VAE. We will soon see why the term autoencoder is also in VAEs.

A natural choice to learn $q(\mathbf{h}|\mathbf{x})$ is to attempt to have it match the true distribution $p(\mathbf{h}|\mathbf{x}, \mathbf{W})$, defined as

$$p(\mathbf{h}|\mathbf{x}, \mathbf{W}) = \frac{p(\mathbf{x}|\mathbf{h}, \mathbf{W})p(\mathbf{h})}{p(\mathbf{x}|\mathbf{W})}$$

We will turn once again to the KL divergence, to provide an objective for this goal: $D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \;||\; p(\cdot|\mathbf{x}, \mathbf{W}))$. We would like to simultaneously minimize this KL, while also minimizing the negative log-likelihood,

$$\operatorname*{argmin}_{\mathbf{W}} -\sum_{i=1}^{n} \ln p(\mathbf{x}_i|\mathbf{W}) + D_{\mathrm{KL}}(q(\cdot|\mathbf{x}_i) \;||\; p(\cdot|\mathbf{x}_i, \mathbf{W}))$$

Let $c_i(\mathbf{W}) = -\ln p(\mathbf{x}_i|\mathbf{W}) + D_{\mathrm{KL}}(q(\cdot|\mathbf{x}_i) \;||\; p(\cdot|\mathbf{x}_i, \mathbf{W}))$ and let us consider just one of these terms c_i in the objective. Notice first that

$$\begin{aligned}
D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \;||\; p(\cdot|\mathbf{x}, \mathbf{W})) &= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x}) - \ln p(\mathbf{h}|\mathbf{x}, \mathbf{W})] \\
&= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}\left[\ln q(\mathbf{h}|\mathbf{x}) - \ln \frac{p(\mathbf{x}|\mathbf{h}, \mathbf{W})p(\mathbf{h})}{p(\mathbf{x}|\mathbf{W})}\right] \\
&= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}\left[\ln q(\mathbf{h}|\mathbf{x}) - \ln p(\mathbf{x}|\mathbf{h}, \mathbf{W}) - \ln p(\mathbf{h}) + \ln p(\mathbf{x}|\mathbf{W})\right] \\
&= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x}) - \ln p(\mathbf{h})] - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})] + \ln p(\mathbf{x}|\mathbf{W})
\end{aligned}$$

where $\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{W})] = \ln p(\mathbf{x}|\mathbf{W})$ because $\ln p(\mathbf{x}|\mathbf{W})$ is a constant wrt $\mathbf{h}$ (and $\mathbb{E}[c] = c$ for a constant c). Additionally, we have that $\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x}) - \ln p(\mathbf{h})] = D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \;||\; p)$ where p is the simple Gaussian over $\mathbf{h}$. Putting this all together, we have that

$$-\ln p(\mathbf{x}|\mathbf{W}) + D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \;||\; p(\cdot|\mathbf{x}, \mathbf{W})) = D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \;||\; p) - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})] \quad (12.1)$$

Lo and behold! We now also have a log-likelihood term for $\mathbf{x}$ that is conditioned on $\mathbf{h}$, with the log inside the expectation: $\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})]$. We did not start out by trying to get this—we were simply trying to find a reasonable sampling distribution for $\mathbf{h}$—but we got this beautiful outcome anyway. Sometimes the universe is kind. This objective is called the negative evidence lower bound (ELBO). This term comes from the fact that the ELBO is equal to $\ln p(\mathbf{x}|\mathbf{W}) - D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \;||\; p(\cdot|\mathbf{x}, \mathbf{W}))$, and so is clearly a lower bound on the *evidence* $\ln p(\mathbf{x}|\mathbf{W})$. Note that we maximize the ELBO, $-D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \;||\; p) + \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})]$ and minimize the negative ELBO. Because we opt to minimize in these notes, we talk about the negative ELBO.

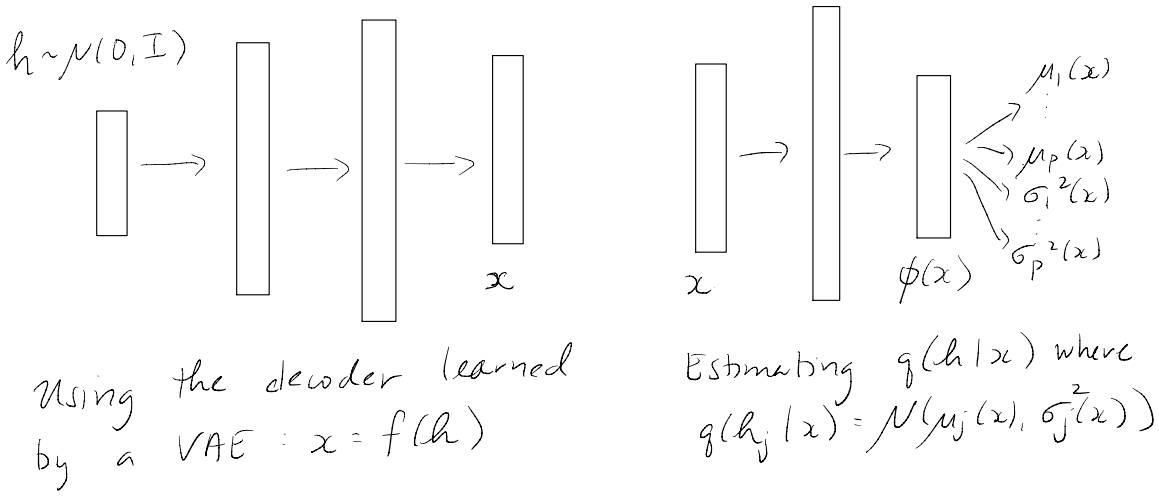


Figure 12.1: The left figure shows how to use the function f learned by the decoder in the Variational Auto-encoder. The right figure shows how to learn $q(\mathbf{h}|\mathbf{x})$ (the encoder) that helps focus sampling to optimize the decoder. Note that a VAE is not like a regular AE, and this analogy can be confusing. Instead we have these two components that are jointly optimized using the negative ELBO.

Remark: Learning $q(\mathbf{h}|\mathbf{x})$ has an additional benefit: simplifying computing the likelihood of new points, when we test our generative model. Once we learn $\mathbf{W}$, we want to be able to compute the likelihood of new data $\tilde{\mathbf{x}}$ under our model $p(\tilde{\mathbf{x}}|\mathbf{W})$. As we discussed above, however, computing this term involves solving an integral. Approximating this integral by sampling $\mathbf{h} \sim p$ requires many many $\mathbf{h}$, since most $p(\mathbf{x}|\mathbf{h}, \mathbf{W})$ is near zero for most $\mathbf{h}$. Instead, we can sample $\mathbf{h}$ from $q(\mathbf{h}|\mathbf{x})$, to more efficiently estimate $p(\tilde{\mathbf{x}}|\mathbf{W})$ when measuring generalization performance.

Now let us move on to optimizing our negative ELBO objective in Equation (12.1). To do so, we need to parameterize $q(\mathbf{h}|\mathbf{x})$. A typical choice is to use $q(\mathbf{h}|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(f_{\mu,\boldsymbol{\theta}}(\mathbf{x}), f_{\sigma,\boldsymbol{\theta}}(\mathbf{x}))$ where $f_{\mu,\boldsymbol{\theta}}(\mathbf{x})$ is a neural network with parameters $\boldsymbol{\theta}$ that outputs a mean estimate $\boldsymbol{\mu} \in \mathbb{R}^d$ and $f_{\sigma,\boldsymbol{\theta}}(\mathbf{x})$ is a neural network with parameters $\boldsymbol{\theta}$ that outputs a diagonal covariance estimate $\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}$, namely it outputs $\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2$. These two terms share parameters $\boldsymbol{\theta}$, because usually they share most of the neural network, as in Figure 12.1. They each have their own separate (private) parameters on the last layer of the neural network. The parameters $\boldsymbol{\theta}$ include all of these parameters, shared and private. Given this last layer—let's call it $\phi(\mathbf{x})$—the mean is a linear function of $\phi(\mathbf{x})$ and the variances $\sigma_j^2(\mathbf{x})$ use a linear function on $\phi(\mathbf{x})$ followed by a transform that ensures the output is positive, such as an exponential. Alternatively, to maintain numerical stability, it is also common for the NN to output the log of $\sigma_j^2(\mathbf{x})$, and use a linear function of $\phi(\mathbf{x})$ to produce this $\ln \sigma_j^2(\mathbf{x})$.

Now we need a mechanism to compute gradients through this new network. The difficulty is that $q(\mathbf{h}|\mathbf{x})$ produces a distribution from which we can sample $\mathbf{h}$; differentiating through such stochastic nodes is not obvious.[1] Fortunately, for our setting, we can take advantage of what is called the *reparameterization trick*. The idea is that we can re-express

[1] Optimizing stochastic neural networks, and the related ideas in stochastic computation graphs, is a very interesting topic, but much beyond what we can cover here.

this stochasticity independent of our parameters, and so compute gradients only on deterministic quantities.

To see why consider what it means to sample $\mathbf{h} \sim q(\cdot|\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}(\mathbf{x}), \boldsymbol{\Sigma}(\mathbf{x}))$ for diagonal $\boldsymbol{\Sigma}(\mathbf{x})$. We are independently sampling $h_j \sim \mathcal{N}(\mu_j(\mathbf{x}), \sigma_j^2(\mathbf{x}))$. This is equivalent to sampling $\epsilon_j \sim \mathcal{N}(0,1)$ and writing $h_j = \mu_j(\mathbf{x}) + \sigma_j(\mathbf{x})\epsilon_j$. Therefore, if we let $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, then we can rewrite

$$\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})] = \mathbb{E}_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[\ln p(\mathbf{x}|h_1 = \mu_1(\mathbf{x}) + \sigma_1(\mathbf{x})\epsilon_1, \ldots, h_p = \mu_p(\mathbf{x}) + \sigma_d(\mathbf{x})\epsilon_p, \mathbf{W})]$$

Now when we compute the gradient for this sample, it can come inside this expectation. For each stochastic gradient descent update, we can sample a point $\mathbf{x}_i$, then sample an $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and easily compute the gradient of $\ln p(\mathbf{x}_i|h_1 = \boldsymbol{\mu}_1(\mathbf{x}_i) + \sigma_1(\mathbf{x}_i)\epsilon_1, \ldots, h_p = \boldsymbol{\mu}_p(\mathbf{x}_i) + \sigma_p(\mathbf{x}_i)\epsilon_p, \mathbf{W})$ with respect to both $\mathbf{W}$ and $\boldsymbol{\theta}$ (which parameterizes $\mu_j(\mathbf{x}_i)$ and $\sigma_j^2(\mathbf{x}_i)$ for all j). We can then use these gradients to update $\mathbf{W}$ and $\boldsymbol{\theta}$.

Notice though that there is one other component in the negative ELBO

$$D_{\mathrm{KL}}(q(\cdot|\mathbf{x}, \boldsymbol{\theta}) \;||\; \mathcal{N}(\mathbf{0}, \mathbf{I})) - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x},\boldsymbol{\theta})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})] \tag{12.2}$$

for $\boldsymbol{\theta}$, which is the KL divergence to the standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$. We can think of this term as a regularizer, that prevents $q(\cdot|\mathbf{x}, \boldsymbol{\theta})$ from becoming too deterministic. This KL term encourages $q(\cdot|\mathbf{x}, \boldsymbol{\theta})$ to stay more similar to a standard Gaussian. It is easy to compute the gradient of this term wrt $\boldsymbol{\theta}$, and this gradient can simply be added to the gradient computed above using reparameterization.

For a more in-depth discussion on VAEs, I highly encourage reading the relatively short tutorial by Carl Doersch [10].

Exercise 39: Simplify the KL divergence between $q(\mathbf{h}|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(f_{\mu,\boldsymbol{\theta}}(\mathbf{x}), f_{\sigma,\boldsymbol{\theta}}(\mathbf{x}))$ and $\mathcal{N}(\mathbf{0}, \mathbf{I})$ and compute the gradient. □

12.3 Connection to Expectation-Maximization

The above approach resembles expectation-maximization (EM). In fact, the ELBO underlies EM as well. The primary distinction is in how we specify $q(\cdot|\mathbf{x})$ and the ability to compute the true expected value. In EM, we set the distribution q over the latent variable to be $p(\mathbf{h}|\mathbf{x}, \boldsymbol{\theta}^{(t)})$ in the E-step. In the maximization step, we can optimize the true expectation $\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}, \mathbf{h}|\mathbf{W})]$ by summing over all the discrete latent $\mathbf{h}$ (which we called $\mathbf{z}$).

In VAEs, we only perform each step approximately. On each update, we modify our parameterized $q(\cdot|\mathbf{x})$ to be closer to

$$p(\mathbf{h}|\mathbf{x}, \mathbf{W}^{(t)}) = \frac{p(\mathbf{x}|\mathbf{h}, \mathbf{W}^{(t)})p(\mathbf{h})}{p(\mathbf{x}|\mathbf{W}^{(t)})}$$

and then we only sample $\mathbf{h}$ to estimate the expectation term $\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})]$. The primary difference is that we do not fully compute the expectation, for the reasons described above: it is expensive. We do not have a closed-form solution, so we resort to sampling. Further, it is intractable to sample $\mathbf{h}$ from $p(\mathbf{h}|\mathbf{x}, \mathbf{W}^{(t)})$, so we approximate it with something that we can easily sample.

To see why the ELBO underlies EM, we can consider again the above equality, in Equation (12.1), but make it slightly more general. Above we simplified just a little bit because $p(\mathbf{h}|\mathbf{W}) = p(\mathbf{h})$ for VAEs. In general, this may not be true, and in fact for mixture models it is not since the distribution over our latent variables depends on $w_1, \ldots, w_m$. Let's use the notation $\boldsymbol{\theta}$ to be all the parameters for our model, and allow $\mathbf{h}$ to be discrete or continuous. Going through similar steps to above,

$$\begin{aligned} D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \mid\mid p(\cdot|\mathbf{x}, \boldsymbol{\theta})) &= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x}) - \ln p(\mathbf{h}|\mathbf{x}, \boldsymbol{\theta})] \\ &= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}\left[\ln q(\mathbf{h}|\mathbf{x}) - \ln \frac{p(\mathbf{x}|\mathbf{h}, \boldsymbol{\theta})p(\mathbf{h}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right] \\ &= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}\left[\ln q(\mathbf{h}|\mathbf{x}) - \ln p(\mathbf{x}|\mathbf{h}, \boldsymbol{\theta}) - \ln p(\mathbf{h}|\boldsymbol{\theta}) + \ln p(\mathbf{x}|\boldsymbol{\theta})\right] \\ &= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x}) - \ln p(\mathbf{h}|\boldsymbol{\theta})] - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \boldsymbol{\theta})] + \ln p(\mathbf{x}|\boldsymbol{\theta}) \end{aligned}$$

The only difference to this same derivation above is that we now have $p(\mathbf{h}|\boldsymbol{\theta})$ rather than $p(\mathbf{h})$. Like before, we can conclude that

$$-\ln p(\mathbf{x}|\boldsymbol{\theta}) + D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \mid\mid p(\cdot|\mathbf{x}, \boldsymbol{\theta})) = D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \mid\mid p(\cdot|\boldsymbol{\theta})) - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \boldsymbol{\theta})] \quad (12.3)$$

which is the more general negative ELBO equation.

In EM, we alternate between optimizing $q(\cdot|\mathbf{x})$ (E-step) and optimizing $\boldsymbol{\theta}$ (M-step). This can be seen as coordinate descent on the above objective.[2] On the E-step, we set $q(\cdot|\mathbf{x}) = p(\cdot|\mathbf{x}, \boldsymbol{\theta})$, which minimizes the KL: the KL becomes zero. On the M-Step, we minimize $-\ln p(\mathbf{x}|\boldsymbol{\theta}) + D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \mid\mid p(\cdot|\mathbf{x}, \boldsymbol{\theta}))$ with a fixed $q(\cdot|\mathbf{x})$. To do so, we can equivalently minimize $D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \mid\mid p(\cdot|\boldsymbol{\theta})) - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})]$. In VAEs, the first term is constant wrt $\boldsymbol{\theta}$, because $p(\mathbf{h}|\boldsymbol{\theta}) = p(\mathbf{h})$, and so we minimized $-\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})]$. For mixture models, however, we cannot drop the first term because it includes $\boldsymbol{\theta}$. Instead, we get that

$$\begin{aligned} &D_{\mathrm{KL}}(q(\cdot|\mathbf{x}) \mid\mid p(\cdot|\boldsymbol{\theta})) - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{W})] \\ &\quad= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x}) - \ln p(\mathbf{h}|\boldsymbol{\theta})] - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \boldsymbol{\theta})] \\ &\quad= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x})] - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}|\mathbf{h}, \boldsymbol{\theta}) + \ln p(\mathbf{h}|\boldsymbol{\theta})] \\ &\quad= \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x})] - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}, \mathbf{h}|\boldsymbol{\theta})] \end{aligned}$$

The first term $\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln q(\mathbf{h}|\mathbf{x})]$ is constant wrt $\boldsymbol{\theta}$, and so we only consider the second term in our minimization. And this is precisely what we did for mixture models: we minimized $-\mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x})}[\ln p(\mathbf{x}, \mathbf{h}|\boldsymbol{\theta})]$ on the M-step.

Therefore, EM can be seen as alternating coordinate descent on the negative ELBO objective, with an unrestricted $q(\cdot|\mathbf{x})$ that can perfectly approximate $p(\cdot|\mathbf{x}, \boldsymbol{\theta})$ for all $\mathbf{x}$ and $\boldsymbol{\theta}$.

12.4 Conditional Generative Models

We can extend both mixture models and VAEs to be conditional distributions: $p(\mathbf{x}|\mathbf{y})$ for some inputs $\mathbf{y}$. Let us start with mixture models. The idea is that the parameters for the

[2] Coordinate descent is an alternative to gradient descent. We can split up our parameters into blocks, say $\mathbf{w}_1$ and $\mathbf{w}_2$ where $\mathbf{w} = [\mathbf{w}_1, \mathbf{w}_2]$. Then we can alternate between computing $\mathbf{w}_1^{(t)} = \mathrm{argmin}_{\mathbf{w}_1} c(\mathbf{w}_1, \mathbf{w}_2^{(t-1)})$ and $\mathbf{w}_2^{(t)} = \mathrm{argmin}_{\mathbf{w}_2} c(\mathbf{w}_1^{(t)}, \mathbf{w}_2)$, until convergence. This is useful in some problems where we can obtain closed-form solutions for parts of the variables. We have exactly this property for EM.

mixture model are now functions of the inputs. For example, for a conditional mixture model with two components modeling $x \in \mathbb{R}$ and conditioned on variables $\mathbf{y}$, we have $p(x|\mathbf{y}) = c_1(\mathbf{y})\mathcal{N}(\mu_1(\mathbf{y}), \sigma_1^2(\mathbf{y})) + c_2(\mathbf{y})\mathcal{N}(\mu_2(\mathbf{y}), \sigma_2^2(\mathbf{y}))$. The coefficients $c_1(\mathbf{y}), c_2(\mathbf{y})$ are now not just scalars, they depend on $\mathbf{y}$. Similarly, the means and variances for the Gaussian components depend on $\mathbf{y}$. We can implement these use any function approximation technique we have seen so far: RBFs, other fixed representations, neural networks, etc.

Let us consider how this looks with neural networks, resulting in what are called *mixture density networks*. The network outputs six values: $c_1(\mathbf{y}), \mu_1(\mathbf{y}), \sigma_1^2(\mathbf{y}), c_2(\mathbf{y}), \mu_2(\mathbf{y}), \sigma_2^2(\mathbf{y})$. They can share the same learned features $\boldsymbol{\phi}(\mathbf{y})$ (last hidden layer) and have outputs

$$\begin{aligned}
c_1(\mathbf{y}) &= \sigma(\boldsymbol{\phi}(\mathbf{y})^\top \mathbf{w}_c) \\
c_2(\mathbf{y}) &= 1 - \sigma(\boldsymbol{\phi}(\mathbf{y})^\top \mathbf{w}_c) \\
\mu_1(\mathbf{y}) &= \boldsymbol{\phi}(\mathbf{x})^\top \mathbf{y}_{\mu,1} \\
\mu_2(\mathbf{y}) &= \boldsymbol{\phi}(\mathbf{x})^\top \mathbf{y}_{\mu,2} \\
\sigma_1^2(\mathbf{y}) &= \exp(\boldsymbol{\phi}(\mathbf{y})^\top \mathbf{w}_{\sigma,1}) \\
\sigma_2^2(\mathbf{y}) &= \exp(\boldsymbol{\phi}(\mathbf{y})^\top \mathbf{w}_{\sigma,2}).
\end{aligned}$$

Notice that $\boldsymbol{\phi}(\mathbf{y})$ could be obtained using any of our data representations, including fixed ones like RBFs. The decision to use fixed representations versus learning them with neural networks is relies on similar reasoning as in the prediction setting. How much data do we have? How much compute and can we use big networks? Do we know how to specify good fixed features? It will be problem specific. And, as before, a common default will be to turn to neural networks, unless there is a good reason to use fixed representations.

Mixture density networks use neural networks, but they do not learn a complex distribution over $\mathbf{x}$ itself. To do so, we turn to conditional VAEs. For a conditional VAE, we want to learn $p(\mathbf{x}|\mathbf{y})$. For example, we may want to generate images of faces $\mathbf{x}$ conditioned on $y \in \{\text{narrow}, \text{wide}\}$, or we might want to generated images of digits $\mathbf{x}$ conditioned on $y \in \{0, 1, 2, \ldots, 9\}$ or $\mathbf{y}$ the one-hot encoding of these digits. With $p(\mathbf{x})$, we can only generate an image of any digit, but with $p(\mathbf{x}|y)$, we can generate an image of a specific digit. For example, we can sample from $p(\mathbf{x}|y = 9)$ to generate images of 9s.

Fortunately, the modification to VAEs to get a conditional VAEs is quite simple. The encoder needs to condition on $\mathbf{y}$ to learn $q(\mathbf{h}|\mathbf{x}, \mathbf{y})$ and the decoder also needs to condition on $\mathbf{y}$ to learn $p(\mathbf{x}|\mathbf{h}, \mathbf{y})$. When we want to use the decoder, we input both $\mathbf{y}$ and a sampled $\mathbf{h} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ to generate $\mathbf{x}$. If we want to sample multiple $\mathbf{x}$ for a given $\mathbf{y}$, then we input the same $\mathbf{y}$ again with a newly sampled $\mathbf{h}$.

For training, we use almost the same loss, the negative ELBO. For a given sample $(\mathbf{x}, \mathbf{y})$, we input both $\mathbf{x}$ and $\mathbf{y}$ into the encoder and input that same $\mathbf{y}$ into the decoder. The negative ELBO for the conditional VAE is

$$D_{\text{KL}}(q(\cdot|\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \;||\; \mathcal{N}(\mathbf{0}, \mathbf{I})) - \mathbb{E}_{\mathbf{h}\sim q(\cdot|\mathbf{x},\mathbf{y},\boldsymbol{\theta})}[\ln p(\mathbf{x}|\mathbf{h}, \mathbf{y}, \mathbf{W})] \tag{12.4}$$

We can use the same reparameterization trick to get the gradient of the second component wrt both $\boldsymbol{\theta}$ and $\mathbf{W}$ and get the gradient of the KL to the standard normal for $\boldsymbol{\theta}$.

Remark: Generative models are different than generative classifiers. We do not consider generative classifiers (e.g., naive Bayes) in these notes, since they are not commonly used, but explain the difference in Appendix A.8.1.

Chapter 13

Evaluating Generative Models

We can estimate generalization error for generative models, just like for predictors. The only difference is in the definition of generalization error. Once we define our measure, and the sample estimator, then we can use all the same strategies (test sets and cross-validation).

One natural choice is the KL divergence between the learned generative model $p(x|\theta)$ and the true generative model $p_{\text{true}}(x)$:

$$D_{\text{KL}}(p_{\text{true}}||p(\cdot|\theta)) = \int_{\mathcal{X}} p_{\text{true}}(x) \ln \frac{p_{\text{true}}(x)}{p(x|\theta)} dx$$

We discussed this divergence[1] in Section 2.5 and showed that

$$D_{\text{KL}}(p_{\text{true}}(x)||p(x|\theta)) = -\mathbb{E}[\ln p(X|\theta)] + \text{a term that depends only on } p_{\text{true}}.$$

Therefore, we can obtain estimates of the generalization error, up to a shift by a constant, simply by estimating the expected log likelihood of data generated by $p_{\text{true}}(x)$, under model $p(x|\theta)$. At this point, we are experts in estimating expectations, simply by using a sample average. Namely, for a given test set $\mathcal{D}_{\text{test}}$ of size m, we have the estimate

$$\text{Perf}(\theta) \stackrel{\text{def}}{=} \frac{1}{m} \sum_{x \in \mathcal{D}_{\text{test}}} \ln p(x|\theta)$$

We say parameters θ are better if they result in models that make the test data more likely, i.e. with higher $\text{Perf}(\theta)$.

With mixture models, it is easy to compute the log likelihood on test data. For a given $\mathbf{x}$, we can compute the likelihood under each component density, by querying the density function, and then weighting these densities with the coefficients. For example, for a mixture models on $x \in \mathbb{R}$ with two Gaussians, $\mathcal{N}(\mu_1, \sigma_1^2)$ and $\mathcal{N}(\mu_2, \sigma_2^2)$ with coefficients $w_1 = 0.2$ and $w_2 = 0.8$, we get likelihood for a point x as

$$p(x) = w_1 (2\pi\sigma_1^2)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2\sigma_1^2}(x-\mu_1)^2\right) + w_2 (2\pi\sigma_2^2)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2\sigma_2^2}(x-\mu_2)^2\right)$$

If $\mu_1 = -1$ and $\sigma_1^2 = 0.3$ and $\mu_2 = 0.4$ and $\sigma_2^2 = 2$, then

$$p(x) = 0.2(2\pi 0.3)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2\times 0.3}(x+1)^2\right) + 0.8(2\pi 2)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2\times 2}(x-0.4)^2\right)$$

and the likelihood of a point $x = -0.5$ is

$$p(-0.5) = 0.2(2\pi 0.3)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2\times 0.3}(-0.5+1)^2\right) + 0.8(2\pi 2)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2\times 2}(-0.5-0.4)^2\right) = 0.4973$$

[1] The KL is not a valid metric (distance) because it is asymmetric and does not satisfy the triangle inequality. It is instead called a divergence.

and the likelihood of a point $x = 0.1$ is

$$p(0.1) = 0.2(2\pi 0.3)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2\times 0.3}(0.1+1)^2\right) + 0.8(2\pi 2)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2\times 2}(0.1-0.4)^2\right) = 1.3253$$

Unfortunately, for VAEs, it is straightforward to generate a sample but not straightforward to get its likelihood. Recall we went to a lot of effort to avoid computing $p(\mathbf{x}|\mathbf{W})$, which is what we would need to get the likelihood. For evaluation, since we only have to do it after training, we might be willing to estimate $p(\mathbf{x}|\mathbf{W})$ by sampling many $\mathbf{h}_1, \ldots, \mathbf{h}_m \sim p(\mathbf{h})$, and using

$$\frac{1}{m}\sum_{k=1}^{m} p(\mathbf{x}_i|\mathbf{h}_k, \mathbf{W}) \approx \mathbb{E}_{\mathbf{h}\sim p}[p(\mathbf{x}|\mathbf{h}, \mathbf{W})] = p(\mathbf{x}_i|\mathbf{W}).$$

We need to do this for every $\mathbf{x}_i$ in our test dataset of size n, with potentially a large number m of samples $\mathbf{h}_k$, requiring mn queries of our decoder. The required number of m can be very large, especially as the dimensionality of $\mathbf{h}$ increases.[2]

We can make this a bit more efficient with out encoder. Recall we used our encoder to focus sampling $\mathbf{h}$ that are likely to have produced $\mathbf{x}$. For evaluation, this could reduce the required size of m. To use our encoder, though, we have to incorporate importance sampling to correct for the fact that we are not sampling from the standard normal. To understand why, notice that

$$\begin{aligned} p(\mathbf{x}_i|\mathbf{W}) = \mathbb{E}_{\mathbf{h}\sim p}[p(\mathbf{x}|\mathbf{h}, \mathbf{W})] &= \int p(\mathbf{x}|\mathbf{h}, \mathbf{W}) p(\mathbf{h}) d\mathbf{h} \\ &= \int \frac{q(\mathbf{h}|\mathbf{x}_i, \boldsymbol{\theta})}{q(\mathbf{h}|\mathbf{x}_i, \boldsymbol{\theta})} p(\mathbf{x}|\mathbf{h}, \mathbf{W}) p(\mathbf{h}) d\mathbf{h} \\ &= \int \frac{p(\mathbf{h})}{q(\mathbf{h}|\mathbf{x}_i, \boldsymbol{\theta})} p(\mathbf{x}|\mathbf{h}, \mathbf{W}) q(\mathbf{h}|\mathbf{x}_i, \boldsymbol{\theta}) d\mathbf{h} \end{aligned}$$

Therefore, we can sample $\mathbf{h} \sim q(\cdot|\mathbf{x}_i, \boldsymbol{\theta})$ to approximate this integral, but we have to correct the distribution with importance sampling ratios $\frac{p(\mathbf{h})}{q(\mathbf{h}|\mathbf{x}_i, \boldsymbol{\theta})}$. Specifically, we sample many $\mathbf{h}_1, \ldots, \mathbf{h}_m \sim q(\cdot|\mathbf{x}_i, \boldsymbol{\theta})$, compute importance sampling ratios $\rho_k \stackrel{\text{def}}{=} \frac{p(\mathbf{h}_k)}{q(\mathbf{h}_k|\mathbf{x}_i, \boldsymbol{\theta})}$ and get estimate

$$\frac{1}{m}\sum_{k=1}^{m} \rho_k p(\mathbf{x}_i|\mathbf{h}_k, \mathbf{W}) \approx p(\mathbf{x}_i|\mathbf{W}).$$

It is also common to qualitatively assess a generative model. The likelihood is one summary statistic that does not adequately represent the utility of a model. It is common to visualize a variety of images generated by the model, to see how realistic and appropriate the images are. There may be images that fail, or systematic oddities (e.g., inability to generate hands). These types of qualitative issues are hard to see through measures like the log likelihood on test data. There is no algorithmic procedure for such qualitative analyses, and it is not necessarily useful for comparing two models rigorously. But, such qualitative analyses are a critical part of assessing our models. Ultimately, you are responsible for what you deploy, so you better understand it with whatever tools you can!

[2] Note that there is a large field of study called numerical integration, exactly geared towards efficiently estimating these kinds of integrals.

Part IV

Advanced Topics

Chapter 14

Dealing with Missing Data

Missing data is ubiquitous in the real world. We say data is missing if for a sample we are missing certain features. For example, one patient may not report their age, whereas another does. If we have a total of d possible input attributes—attributes like age and blood type–then the corresponding inputs $\mathbf{x}_1$ may only have a subset of those possible attributes available, due to such data collection issues. Another setting where this might occur is that older data was collected asking patients for less information, and newer patient records have additional attributes.

In this chapter, we see how to leverage some of the ideas we have seen on representations, particularly latent variable methods, to handle missing data. We focus on the case where the data is missing from the conditioned variable—inputs—rather than the targets. For an input $\mathbf{x}_i$, we assume $m_i \in \{0, 1, \ldots, d-1\}$ attributes are missing, where each data point i may have a different number m_i of attributes missing. We allow for $m_i = 0$, since no attributes may be missing, and do not allow $m_i = d$, since then the input vector has no information. We let $\mathcal{M}_i$ be the set of indices for the missing attributes, for the i-th input, and $\mathcal{A}_i$ the set of available indices, namely with $\mathcal{A}_i = \{1, 2, \ldots, d\} \backslash \mathcal{M}_i$.

14.1 Imputation: Filling in Missing Values

Imputation means that we *impute* or *fill in* the missing values, given the observed variables. We have already seen an instance of imputation: supervised learning. We can think of the targets as the commonly unobserved variables that we need to infer, given the other variables (features). This is a very specific missing data setting, in the sense that we have assumed that these variables are not missing during training, and only that single variable is missing in deployment.

Slightly more general is the *semi-supervised learning* setting, where in training we only have label (targets) for a subset of the training set. We want to impute the missing labels for the rest of the training set. A related idea is *transductive learning*, where we take both the training and test datasets, and impute the missing labels for the test dataset, as shown in Figure 14.1.

More generally, we can imagine that our input data is peppered with missing values, rather than being concentrated on only one feature. Our goal, nonetheless, remains to fill in those missing values. This problem setting is called *matrix completion*, and one standard approach has been to use matrix factorization (methods like PCA). The idea is that if we can identify the latent factors $\mathbf{h}$ for an input $\mathbf{x}$, even if that input $\mathbf{x}$ has missing values, then we can complete $\mathbf{x}$ by using the predicted $\hat{\mathbf{x}}$ from the latent factors: $\hat{\mathbf{x}} = \mathbf{hD}$. We replace just the missing parts of $\mathbf{x}$ with the corresponding entries in $\hat{\mathbf{x}}$. Namely, $\mathbf{x}(\mathcal{M}_i) = \hat{\mathbf{x}}(\mathcal{M}_i)$.

Now the question is how it is that we obtain latent factors under missing data. The

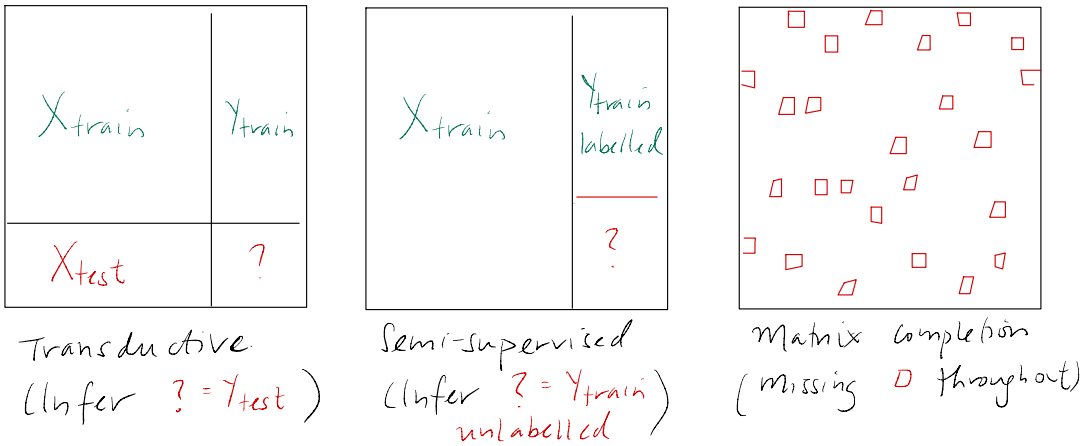


Figure 14.1: Missing Data settings. Semi-supervised learning and transductive learning just consider missing targets, and infer those targets during learning. Matrix completion is about imputing any missing features, not just ones grouped in the last columns (the columns corresponding to what we usually call targets).

matrix factorization optimization for PCA makes this relatively straightforward. Recall that for complete data, for each input $\mathbf{x}_i$, we minimize

$$\|\mathbf{x}_i - \mathbf{h}_i\mathbf{D}\|_2^2 = \sum_{j=1}^{d}(x_{ij} - \mathbf{h}_i\mathbf{D}_{:,j})^2 = \sum_{j=1}^{d}(x_{ij} - \hat{x}_{ij})^2 \qquad \triangleright \text{ for } \hat{\mathbf{x}}_i = \mathbf{h}_i\mathbf{D}.$$

If parts of $\mathbf{x}_i$ are missing, we can simply try to find the best $\mathbf{h}_i$ based on the available information at the indices $\mathcal{A}_i$

$$\sum_{j\in\mathcal{A}_i}(x_{ij} - \hat{x}_{ij})^2 \qquad \triangleright \text{ for } \hat{\mathbf{x}}_i = \mathbf{h}_i\mathbf{D}.$$

The training objective, therefore, for our dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathcal{A}_i)\}$ with vectors $\mathbf{x}_i$ with available values at indices $\mathcal{A}_i$ (and other indices missing) is

$$\min_{\mathbf{h}_1,\ldots,\mathbf{h}_n\in\mathbb{R}^p,\mathbf{D}\in\mathbb{R}^{p\times d}} \sum_{i=1}^{n}\sum_{j\in\mathcal{A}_i}(x_{ij} - \mathbf{h}_i\mathbf{D}_{:,j})^2 \tag{14.1}$$

This optimization may seem underconstrained, because the loss is now a subset of the squared error previously considered. But it is not underconstrained because $\mathbf{D}$ is shared across all datapoints, $\mathbf{h}_i$ is shared across features for $\mathbf{x}_i$ and $p < d$. Every datapoint $\mathbf{x}_i$ that has feature j available has to share the same $\mathbf{D}_{:,j}$ to reconstruct x_{ij}. Further, we have to use the same $\mathbf{h}_i$ for datapoint i, regardless of which feature j we are reconstructing. For example, we can easily set $\mathbf{h}_i$ to get $\hat{x}_{i1} = \mathbf{h}_i\mathbf{D}_{:,1}$, but then we also have to use $\mathbf{h}_i$ to get $x_{i2} = \mathbf{h}_i\mathbf{D}_{:,2}$ and $x_{i4} = \mathbf{h}_i\mathbf{D}_{:,4}$. We have to pick an $\mathbf{h}_i$ that works for all these available indices. This sharing in $\mathbf{D}$ and $\mathbf{h}_i$ sufficiently constrains the optimization that we can have quite a bit of missing information and still get a unique reconstruction of the data matrix.

Example 17: A canonical example of matrix completion is for movie rankings. Each user $i \in \{1, 2, \ldots, n\}$ has an associated list of movie rankings $\mathbf{x}_i \in \mathbb{R}^d$ for the total set of d

movies. Of course, no user has ranked every movie, and so naturally $\mathbf{x}_i$ has (many) missing entries. Or, in other words, user i only has rankings for a subset of movies with indices $\mathcal{A}_i$.

When we solve for $\mathbf{h}_i$ and $\mathbf{D}$ in Equation (14.1), we can think about what each of these encodes. The hidden dimension of size p can be seen as latent factors that explain why users like movies. For example, let us imagine that hidden dimension k corresponds to `Movies that are Happy`. $\mathbf{D}_{k,:} \in \mathbb{R}^d$ can be seen as a canonical list of moving ratings based solely on how Happy they are: high rankings for Happy movies and low rankings for Unhappy movies. The element h_{ik} for user i corresponds to how much they like Happy movies. If they like Happy movies, then h_{ik} should be positive and the vector of rankings $h_{ik}\mathbf{D}_{k,:}$ helps explain why user i has higher ratings for happy movies in $\mathbf{x}_i$. Other dimensions could correspond to factors like `Long Movies` or `Cult Movies`.[1]

For a given user with rankings $\mathbf{x}_i$, with the available movie rankings $\mathcal{A}_i$, these available rankings help us identify $\mathbf{h}_i$: the properties of the user, like if they like Happy movies and/or Cult movies. Given these latent factors, we can then fill in the remaining movie rankings for the missing set $\mathcal{M}_i$ using: $\mathbf{x}_{ij} = \mathbf{h}_i\mathbf{D}_{:j}$ for $j \in \mathcal{M}_i$. Essentially, the optimization uses knowledge about how other users have ranked movies, and relationships between movies that we can infer solely based on how often they are ranked similarly (or oppositely). □

14.2 Imputation of Missing Data for Prediction

The above matrix completion example is an instance where inferring the features was useful to make recommendations to a user. The goal was completing the matrix, without a separate classification or regression goal. This contrasts methods that complete the input data, so that they can then apply standard regression and classification algorithms. For example, for predicting whether a patient has a disease, we might take a two stage process. Step 1: Impute any missing attribute information about the patient (e.g., age), to obtain complete data. Step 2: Use a standard classifier or regressor on this completed data.

We can use the imputation approach in the previous section for this two stage approach. We are using PCA to infer missing values, by presuming there is a latent structure to exploit. Once we have those missing values, then it is easy to apply any regression approach. It is not uncommon to fill in missing values using a simple method like PCA, and then use that completed data to learn a model with a neural network. With PCA, we can do a one-shot approach to fill in missing values. For more complex models, like autoencoders and VAEs, we need to take an iterated approach.

To understand why we need an iterated approach, we can consider what optimization problem we are solving when we have missing variables. We treat the missing variables as unknowns, and optimize over them as well. The typical negative log likehood minimization $\min_{\mathbf{W}} \sum_{i=1}^n \ln p(\mathbf{x}|\mathbf{W})$ becomes instead

$$\min_{\mathbf{x}_{1,\mathcal{M}_1},\ldots,\mathbf{x}_{n,\mathcal{M}_n}} \min_{\mathbf{W}} \sum_{i=1}^{n} -\ln p(\mathbf{x}_{i,\mathcal{A}_i}, \mathbf{x}_{i,\mathcal{M}_i}|\mathbf{W}) \tag{14.2}$$

[1] Of course, all of this is made up. It is hard to say exactly what the latent dimensions actually correspond to. But, matrix completion really has been used for this problem, and it worked surprisingly well! Netflix put out a competition years ago, and a winning entry used matrix completion to fill in rankings.

where $\mathbf{x}_{i,\mathcal{A}_i}, \mathbf{x}_{i,\mathcal{M}_i}$ gives us the complete $\mathbf{x}$. For a given choice of $\mathbf{x}_{i,\mathcal{M}_i}$ in the outer optimization, we have a complete $\mathbf{x}$ and we can use a standard update to the weights to minimize the negative log likelihood.

Let us make this concrete by optimizing Equation (14.2) for an autoencoder. We have to start by initializing our variables, as usual. A typical choice is to initialize $\mathbf{x}_{i,\mathcal{M}_i}$ to zero, or the mean value of that feature computed in the training dataset. We can initialize $\mathbf{W}$ using a standard initialization for neural networks, like He initialization. The loss for the autoencoder for one sample is $\|\mathbf{x}_i - \hat{\mathbf{x}}_i\|_2^2$, where $\hat{\mathbf{x}}_i$ is the output of the autoencoder given input $\mathbf{x}_i$. Note here that $\mathbf{x}_i$ consists of $\mathbf{x}_{i,\mathcal{A}_i}$ and the current estimate of the missing values $\mathbf{x}_{i,\mathcal{M}_i}$, so it is a completed datapoint without any missingness. We already know how to get the gradient of the weights $\mathbf{W}$ for this loss: we simply use backprop with this loss $\|\mathbf{x}_i - \hat{\mathbf{x}}_i\|_2^2$. For the outer optimization over $\mathbf{x}_{i,\mathcal{M}_i}$, we could use gradient descent, but here we have a simple closed-form solution: setting $x_{i,j} = \hat{x}_{i,j}$ for each $j \in \mathcal{M}_i$. This choice actually gives us zero error.

Overall, the procedure involves iteratively doing gradient descent updates for $\mathbf{W}$ followed by setting $\mathbf{x}_{i,\mathcal{M}_i} = \hat{\mathbf{x}}_{i,\mathcal{M}_i}$, which will eventually converge to a stationary point. A specific instance of this procedure could consist of steps

1. Initialize $\mathbf{W}$ use He initialization. Initialize $\mathbf{x}_{i,\mathcal{M}_i}$ to zero for all i, in other words we are using zero imputation to create a completed dataset. Note that these $\mathbf{x}_{i,\mathcal{M}_i}$ will be updated, in place, inside our data matrix.

2. Grab a mini-batch $\mathcal{B}$ and

   (a) Compute the forward pass on $\mathcal{B}$ to get the output $\hat{\mathbf{x}}_i$ for each $i \in \mathcal{B}$.

   (b) Compute the gradients for the mini-batch using backprop and update $\mathbf{W}$.

   (c) Update $\mathbf{x}_{i,\mathcal{M}_i} = \hat{\mathbf{x}}_{i,\mathcal{M}_i}$ in our data matrix for each $i \in \mathcal{B}$.

3. Repeat Step 2 until both $\mathbf{W}$ and $\mathbf{x}_{1,\mathcal{M}_1}, \ldots, \mathbf{x}_{n,\mathcal{M}_n}$ have stopped changing, within some tolerance, or once you hit a maximum number of iterations.

Once we converge, we both have the parameters for the model as well as estimates for the missing values in the dataset.[2]

We can also use this idea to fill in missing values for a new data point $\mathbf{x}$, after training. In this case, $\mathbf{W}$ is fixed, and we can query our autoencoder $f_{\mathbf{W}}(\mathbf{x})$ on (complete) inputs $\mathbf{x}$ to produce output $\hat{\mathbf{x}}$. To get such a complete $\mathbf{x}$, we solve

$$\min_{\mathbf{x}_{\mathcal{M}}} \|\mathbf{x} - f_{\mathbf{W}}(\mathbf{x})\|_2^2 \qquad \text{where } \mathbf{x} \text{ consists of } \mathbf{x}_{\mathcal{A}} \text{ and } \mathbf{x}_{\mathcal{M}}$$

The procedure for this is a subset of the above

[2]This iterative imputation is called multivariate iterative chained equations (MICE) or fully conditional specification. This name comes from the fact that we predict the missing values by conditioning on the other variables. MICE is often implemented a bit differently from above, as the typical strategy is to predict each feature one at a time. The algorithms cycle through the features, predicting the values for that feature to fill it in, and then using those as the new values. Predicting them all at once, however, is straightforward with a neural network (an autoencoder), and reflects the same idea, so we opt for that approach here. And it should actually be better to predict them all at once, because using this cyclic or chained strategy is like using coordinate descent rather than jointly optimizing all variables at once. Coordinate descent is typically slower, and if we can optimize all variables at once, we typically should.

1. Initialize $\mathbf{x}_{\mathcal{M}}$ to zero.
2. Compute the forward pass to get the output: $\hat{\mathbf{x}} \leftarrow f_{\mathbf{W}}(\mathbf{x})$.
3. Update $\mathbf{x}_{\mathcal{M}} = \hat{\mathbf{x}}_{\mathcal{M}}$.
4. Repeat Steps 2 and 3 until $\mathbf{x}_{\mathcal{M}}$ stops changing, within some tolerance or once you hit a maximum number of iterations (e.g., it is often enough to use 10 or 20).

This procedure hardly feels like an optimization, since we are just iteratively querying the autoencoder for the new outputs and feeding in these as new inputs. But, implicitly, we are solving the above optimization.

In summary, this approach involves first imputing missing data (e.g., using PCA or an autoencoder) and then handing that complete data to your favorite regression or classification algorithm. This two stage approach is conceptually simple, but has disadvantages. One disadvantage of imputation for missing data, if we are just doing prediction, is that we may be solving a harder problem than necessary. It is hard to know what the missing values should have been, and for prediction we really only needed $p(y|\mathbf{x})$, not $\mathbf{x}$ itself. Additionally, this two stage approach fills in the missing values without considering how those values improve prediction accuracy. In fact, in some cases, we can do a direct approach, where we predict y without trying to get accurate estimates of missing values.

1. **Two Stage Approach**. Stage 1: Impute the missing values. Stage 2: Use the complete data for prediction. This is what we did in this section, Section 14.2.
2. **Direct Approach**: Learn a classifier (or regressor) that naturally handles missing data. We discuss this approach in the next section, Section 14.3.

Exercise 40: Consider how we could extend the above algorithm with autoencoders, to supervised autoencoders, to jointly complete the data and a learn a predictor. Write down the optimization and the procedure. □

Remark: We only considered a single imputation above, trying to find the best completion of the missing data. We could instead maintain a distribution over possible missing values, and sample multiple imputations. This is similar to how in PCA we consider the single best $\mathbf{h}$ and in PPCA (and VAEs), we reasoned about a distribution of $\mathbf{h}$. Multiple imputation involves producing different plausible completions, resulting in multiple possible dataset (e.g., 30), on which we then run regression. We get 30 different predictions for each datapoint, and use these 30 different predictions to create an interval around our predictions. We do not cover multiple imputation here, but for a bit more information see Appendix A.10.1.

14.3 Direct Methods for Prediction Under Missing Data

In this section we discuss one simple approach to directly predict $p(y|\mathbf{x})$, without first imputing missing values in $\mathbf{x}$. Our goal is to learn a function conditioned on all the available data: the features that are available and information about which features are missing, $f(\mathbf{x}_{\mathcal{A}}, \mathcal{M})$. We should be able to learn a more accurate function this way, compared to first imputing and then learning as usual on the imputed data. The two stage approach

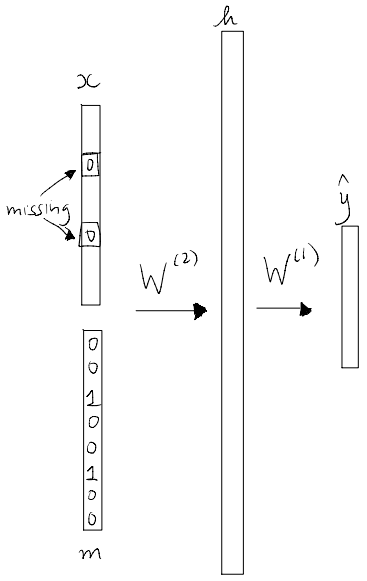


Figure 14.2: Direct prediction for a neural network on data with missingness. The missing values in $\mathbf{x}$ *are set to zero and an indicator vector for which features are missing also inputted into the network. This network learns* $f_{\mathbf{W}}(\mathbf{x}, \mathbf{m}) = \hat{y}$.

has bias, depending on the imputation, whereas $f(\mathbf{x}_{\mathcal{A}}, \mathcal{M})$ is conditioned only on the (true) given data. The primary consideration now is what function class to use to effectively learn $f(\mathbf{x}_{\mathcal{A}}, \mathcal{M})$.

If we want to learn a neural network f, then a simple but typically effective choice is to set the elements $\mathbf{x}_{\mathcal{M}_i} = 0$ and additionally input $\mathcal{M}$ as an indicator vector. This vector $\mathbf{m}_i \in \mathbb{R}^d$ corresponds to a binary vector where there is a 1 for position j if the feature is missing and 0 if it is available. You can think of this almost like an `if` statement. If x_j is available, then use the weights on x_j to produce the features in the first layer. If x_j is not available, then the weights on x_j multiply by zero and do not add to the sum. Instead, the weights on the indicator are used. This approach is depicted in Figure 14.2.

To see more precisely why, let's consider a simpler linear setting for a target $y \in \mathbb{R}$. We have weights $\mathbf{w}_x \in \mathbb{R}^{d \times 1}$ on $\mathbf{x} \in \mathbb{R}^{1 \times d}$ and $\mathbf{w}_m \in \mathbb{R}^{d \times 1}$ on $\mathbf{m} \in \mathbb{R}^{1 \times d}$, where jointly we actually stack these all and use $\hat{y} = [\mathbf{x}, \mathbf{m}]\mathbf{w}$ for $\mathbf{w} = \begin{bmatrix} \mathbf{w}_x \\ \mathbf{w}_m \end{bmatrix} \in \mathbb{R}^{2d \times 1}$.Then notice that

$$[\mathbf{x}, \mathbf{m}]\mathbf{w} = \sum_{j \in \mathcal{A}_i} x_j w_{x,j} + \sum_{j \in \mathcal{M}_i} w_{m,j}$$

We still have a linear function on the available x_j and the missingness indicator just allows us to shift the prediction by $w_{m,j}$ for any missing features. This seems quite limited. However, once we add depth, recall that we can start to `and` input features. In the first layer, this shift can be different for each hidden node, allowing us as before to learn different partitionings. The next layer can start to `and` these partitionings, and effectively learn different predictors for different parts of the input space. The neural network can input these two stacked vectors and learn a complex function that conditions on both available features and the missingness pattern.

Remark: You may be wondering how the terms MCAR, MAR and MNAR did not come up in a chapter about missing data! I think these terms are focused on too much, especially given we almost always have data that is missing not at random (MNAR). The techniques in this chapter make no assumptions on whether data is missing at random, so we do not need to reason about these cases. For the interested reader, you can see Appendix A.10.1 for a little bit of information on why people care about these terms.

Chapter 15

Uncertainty Estimation and Bayesian Approaches

The goal of Bayesian methods is to maintain the posterior distribution, $p(\mathbf{w}|\mathcal{D})$. These weights $\mathbf{w}$ can be the parameters for a prediction function, like the weights in linear regression, or the weights for a generative model. In MAP, we use the point estimate $\text{argmax}_{\mathbf{w}}\, p(\mathbf{w}|\mathcal{D})$. Bayesian methods, therefore, estimate more information: they allow us to reason about our certainty in our weights. If the distribution is wide—has high variance—then we have low certainty. With more data, the posterior concentrates and in the limit we again obtain a point estimate.

In this chapter we first review Bayesian linear regression, one of the simplest Bayesian approaches. This involves maintaining the posterior assuming the prior $p(\mathbf{w})$ is Gaussian and $p(y|\mathbf{x})$ is Gaussian with mean $\mathbf{xw}$ and unknown variance σ_y. The posterior is maintained over both $(\mathbf{w}, \sigma_y)$, and corresponds to a normal-inverse Gamma distribution. The beauty of Bayesian linear regression is that it has a nice closed form due to having a conjugate prior. It is, however, restricted to these assumptions and linear regression.

Just like when we moved beyond linear regression to the nonlinear setting with data representations, we can extend to more general Bayesian methods using data representations like kernels and neural networks. We examine one of the most widely used Bayesian approaches, that rely on kernels: Gaussian Processes.

15.1 Bayesian Linear Regression

Bayesian estimation involves maintaining the entire posterior distribution, $p(\mathbf{w}|\mathcal{D})$. Once we have looked at MAP, the extension to Bayesian estimation is not a big leap. For MAP, we already had to specify a prior to obtain $\text{argmax}_{\mathbf{w}\in\mathcal{F}}\, p(\mathbf{w}|\mathcal{D})$. For Bayesian estimation, we need to maintain the entire posterior $p(\mathbf{w}|\mathcal{D})$, not just the mode. We simplify the explanation by only considering the univariate case: $w \in \mathbb{R}$.

Assume that $p(y|x) = \mathcal{N}(\mu = xw, {\sigma_y}^2)$ for some fixed $\sigma_y \in \mathbb{R}$. This is the assumption we made for linear regression, and then for MAP with a Gaussian prior on the weights. Again, let's assume a Gaussian prior on the weights $p(w) = \mathcal{N}(0, {\sigma_y}^2/\lambda)$ for some (regularization) parameter $\lambda > 0$. Then we get

$$
\begin{aligned}
p(w|\mathcal{D}) &= \frac{p(\mathcal{D}|w)p(w)}{p(\mathcal{D})} && \triangleright\, p(x_i, y_i|w) = p(y_i|x_i, w)p(x_i|w) = p(y_i|x_i, w)p(x_i) \\
&= \frac{p(w)\prod_{i=1}^n p(y_i|x_i, w)p(x_i)}{\int p(w)\prod_{i=1}^n p(y_i|x_i, w)p(x_i)dw} && \triangleright\, \prod_{i=1}^n p(x_i) \text{ cancels in numerator/denominator} \\
&= \frac{p(w)\prod_{i=1}^n p(y_i|x_i, w)}{\int p(w)\prod_{i=1}^n p(y_i|x_i, w)dw}
\end{aligned}
$$

Computing the posterior is complicated by the integral in the denominator. In some cases, though, this integral can be solved analytically, and the posterior has a simple known form. This was the case with *conjugate priors*. For a given $p(y|x,w)$, a conjugate prior $p(w)$ is one where the posterior $p(w|\mathcal{D})$ is of the same form as the prior (example, both Gaussian).

For Bayesian linear regression, where $p(y|x) = \mathcal{N}(\mu = xw, {\sigma_y}^2)$, the conjugate prior is in fact the prior used for ℓ_2 regularization: $p(w) = \mathcal{N}(0, {\sigma_y}^2/\lambda)$. Given this prior, with prior mean $\mu_0 = 0$ and prior variance $\sigma_0^2 = {\sigma_y}^2/\lambda$, it can be derived that

$$p(w|\mathcal{D}) = \mathcal{N}(\mu_n, \sigma_n^2) \qquad \text{where} \quad \sigma_n^2 = \frac{{\sigma_y}^2}{\sum_{i=1}^n x_i^2 + \lambda}$$
$$\mu_n = \frac{\sum_{i=1}^n x_i y_i + \lambda\mu_0}{\sum_{i=1}^n x_i^2 + \lambda} = \frac{\sum_{i=1}^n x_i y_i}{\sum_{i=1}^n x_i^2 + \lambda} = \frac{\sigma_n^2}{{\sigma_y}^2}\sum_{i=1}^n x_i y_i$$

The MAP solution corresponds to the mode of this distribution: μ_n. Additionally, we can obtain a credible interval for which weights are plausible given the data, based on the variance σ_n. If the variance is big, then even after seeing the data there are many plausible values for w. As n gets larger, notice that σ_n^2 shrinks.

We can similarly obtain the posterior if we have multivariate inputs. Let us assume that we take $\boldsymbol{\mu}_0 = 0$, which is a typical choice. Then we have that

$$p(\mathbf{w}|\mathcal{D}) = \mathcal{N}(\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n) \qquad \text{where} \qquad \boldsymbol{\Lambda} = \sum_{i=1}^n \mathbf{x}_i^\top \mathbf{x}_i + \lambda\mathbf{I} \tag{15.1}$$
$$\boldsymbol{\Sigma}_n = {\sigma_y}^2 \boldsymbol{\Lambda}^{-1}$$
$$\boldsymbol{\mu}_n = \boldsymbol{\Lambda}^{-1}\sum_{i=1}^n \mathbf{x}_i^\top y_i$$

For linear regression, though, we typically do not know the variance ${\sigma_y}^2$. Fortunately, even when extending more generally to this setting, we have a conjugate prior. First consider the univariate case. We need now a prior on weights $w \in \mathbb{R}$ and also the variance ${\sigma_y}^2$. The conjugate prior is called the Normal-Inverse-Gamma (NIG) distribution, which has four parameters: $\mu_n, \lambda_n, a_n, b_n$. For prior parameters $\mu_0 \in \mathbb{R}$ and $\lambda_0, a_0, b_0 > 0$ (e.g., $\mu_0 = 0, \lambda_0 = 0.1, a_0 = 3, b_0 = 10$), we get posterior

$$p(w, {\sigma_y}^2|\mathcal{D}) = \text{NIG}(\mu_n, \lambda_n, a_n, b_n) \qquad \text{where} \quad \lambda_n = \sum_{i=1}^n x_i^2 + \lambda_0$$
$$\mu_n = \frac{\sum_{i=1}^n x_i y_i + \lambda_0\mu_0}{\sum_{i=1}^n x_i^2 + \lambda_0} = \frac{\sum_{i=1}^n x_i y_i + \lambda_0\mu_0}{\lambda_n}$$
$$a_n = a_0 + \tfrac{1}{2}n$$
$$b_n = b_0 + \tfrac{1}{2}\left(\sum_{i=1}^n y_i^2 + \lambda_0\mu_0^2 - \lambda_n\mu_n^2\right)$$

Notice that $p(w, {\sigma_y}^2) = p(w|{\sigma_y}^2)p({\sigma_y}^2)$. A key property of an NIG distribution $p(w, {\sigma_y}^2)$ with parameters μ, λ, a, b is that $p(w|{\sigma_y}^2)$ is Gaussian $\mathcal{N}(\mu, {\sigma_y}^2/\lambda)$ and $p({\sigma_y}^2)$ is an inverse gamma distribution with parameters a, b. For the NIG, the mode of the distribution is $\mathbb{E}[(w, {\sigma_y}^2)] = (\mu_n, \frac{b_n}{a_n - 1})$. The solution for w is the same as for MAP above. And now we also have an estimate for the most likely value for the variance of the noise $\frac{b_n}{a_n - 1}$.

We can use this distribution to reason about a plausible set of values for the weights, called the credible interval. The variance of the weights, under the NIG, corresponds to $\frac{b_n}{(a_n-1)\lambda_n}$ for $a_n > 1$. If this term is large, then the set of plausible weights are large. We can be more precise by computing $p(w \in [a, b]|\mathcal{D}) = 0.95$ to get a 95% credible interval for w. We can compute the marginal for w, of the NIG: it is a Student's t-distribution, with mean μ_n, scale parameter $\frac{a_n}{b_n\lambda_n}$ and degrees of freedom $2a_n$. Consequently, we can get a 95% credible interval using $[\mu_n - \epsilon, \mu_n + \epsilon]$ for $\epsilon = t_{0.025,2a_n}\frac{a_n}{b_n\lambda_n}$.

All the above updates can be extended to the multivariate case. We need now a prior on the vector of weights $\mathbf{w} \in \mathbb{R}^d$ and the variance ${\sigma_y}^2$, where the variance is still a scalar since it is the variance for scalar y given $\mathbf{x}$. The conjugate prior is still the Normal-Inverse-Gamma (NIG) distribution, but now for the multivariate case, with four parameters: $\boldsymbol{\mu}_n \in \mathbb{R}^d, \boldsymbol{\Lambda}_n \in \mathbb{R}^{d\times d}, a_n, b_n$. For prior parameters $\boldsymbol{\mu}_0 \in \mathbb{R}^d$, $\boldsymbol{\Lambda}_0 \succeq 0$ and $a_0, b_0 > 0$ (e.g., $\boldsymbol{\mu}_0 = 0, \boldsymbol{\Lambda}_0 = \mathbf{I}, a_0 = 3, b_0 = 10$), we get posterior $p(\mathbf{w}, {\sigma_y}^2|\mathcal{D}) = \text{NIG}(\boldsymbol{\mu}_n, \boldsymbol{\Lambda}_n, a_n, b_n)$ where

$$\begin{aligned}
\boldsymbol{\Lambda}_n &= \sum_{i=1}^{n} \mathbf{x}_i^\top \mathbf{x}_i + \boldsymbol{\Lambda}_0 = \mathbf{X}^\top\mathbf{X} + \boldsymbol{\Lambda}_0 \\
\boldsymbol{\mu}_n &= \boldsymbol{\Lambda}_n^{-1}\left(\sum_{i=1}^{n} \mathbf{x}_i^\top y_i + \boldsymbol{\Lambda}_0\boldsymbol{\mu}_0\right) = \boldsymbol{\Lambda}_n^{-1}\left(\mathbf{X}^\top\mathbf{y} + \boldsymbol{\Lambda}_0\boldsymbol{\mu}_0\right) \\
a_n &= a_0 + \tfrac{1}{2}n \\
b_n &= b_0 + \tfrac{1}{2}\left(\sum_{i=1}^{n} y_i^2 + \boldsymbol{\mu}_0^\top\boldsymbol{\Lambda}_0\boldsymbol{\mu}_0 - \boldsymbol{\mu}_n^\top\boldsymbol{\Lambda}_n\boldsymbol{\mu}_n\right)
\end{aligned} \tag{15.2}$$

We can again use this distribution to reason about the credible region over $\mathbf{w}$. The covariance of the weights, under the NIG, corresponds to $\frac{b_n}{a_n-1}\boldsymbol{\Lambda}_n^{-1}$ for $a_n > 1$. We can compute $p(\mathbf{w} \in [\mathbf{a}, \mathbf{b}]|\mathcal{D}) = 0.95$ to get a 95% credible region for $\mathbf{w}$. To do so, we need the marginal for $\mathbf{w}$, of the NIG: it is a Student's t-distribution, with mean $\boldsymbol{\mu}_n$, scale parameter $\frac{a_n}{b_n}\boldsymbol{\Lambda}_n^{-1}$ and degrees of freedom $2a_n$. Obtaining the credible region is a bit more complicated in this multivariate space, and we will not explicitly need it. Instead, we will want to reason about uncertainty in our predictions, as we discuss in the next section.

15.2 Using the Bayesian Posterior over Weights

Our goal is to obtain credible intervals around predictions, not just around the weights. Namely, given $p(\mathbf{w}|\mathcal{D})$, we would like to reason about $p(f(\mathbf{x})|\mathcal{D})$ where $f(\mathbf{x}) = \mathbf{x}\mathbf{w}$ for any $\mathbf{x}$. Notice this is not about the stochasticity in y: it is not about computing $p(y|\mathbf{x})$. Rather, it is about our uncertainty in the our prediction $f(\mathbf{x}) \approx \mathbb{E}[Y|\mathbf{x}]$. Before moving on to obtaining this credible interval, let us reason a bit more about what uncertainty is being characterized.

In Bayesian linear regression we are trying to identify the best linear function in our set of linear functions $\mathcal{F}$. We know that linear regression provides the global minimum of the linear regression objective. Therefore, as we get more and more samples (n gets bigger), the function obtained by linear regression gets closer to the best linear function in $\mathcal{F}$, in terms of minimizing the squared errors across the entire space. For any given point $\mathbf{x}$ for which we make a prediction $f(\mathbf{x})$, and then an actual outcome y is revealed, we can reason about three sources of error. Let $f_{\text{best}} \in \mathcal{F}$ be the best linear function and f^* the true $\mathbb{E}[Y|x]$,

which may not be in $\mathcal{F}$. Then

$$f(\mathbf{x}) - y = \underbrace{f(\mathbf{x}) - f_{\text{best}}(\mathbf{x})}_{\text{due to insufficient data}} + \underbrace{f_{\text{best}}(\mathbf{x}) - f^*(\mathbf{x})}_{\text{bias}} + \underbrace{f^*(\mathbf{x}) - y}_{\text{irreducible error}}$$

Our credible interval is only reasoning about the uncertainty due to the first term: $f(\mathbf{x}) - f_{\text{best}}(\mathbf{x})$. We can map this to the variance component of reducible error of the generalization error. Recall that the GE decomposed into reducible error and irreducible error, and reducible error further decomposed into bias and variance. The irreducible error is due to the variance in Y given x: due to ${\sigma_y}^2$. When we make a prediction $f(\mathbf{x})$, even if it is the true function $f^*(\mathbf{x})$, we will always have some uncertainty in the accuracy of the prediction because the outcome is stochastic. This uncertainty is sometimes called *aleatoric uncertainty*. The uncertainty estimates given by Bayesian linear regression are instead about the reducible error, and more specifically about the variance component. As we get more and more data, the variance term gets smaller, because the set of plausible functions for the larger dataset becomes smaller. Across different large datasets, we will see relatively consistent functions. This uncertainty estimate is often called *epistemic uncertainty*.

Now let us return to reasoning about our uncertainty in $f(\mathbf{x})$, due to insufficient data. We use the result that a linear weighting of Gaussian variables is again Gaussian. Let $\mathbf{v}$ be a d-dimensional multivariate Gaussian vector $\mathbf{v}$ with mean $\boldsymbol{\mu} \in \mathbb{R}^d$ and covariance $\boldsymbol{\Sigma} \in \mathbb{R}^d$. Let $\mathbf{b} \in \mathbb{R}^d$ be any vector of coefficients. Then we know that the univariate random variable z resulting from the linear weighting of $\mathbf{v}$, $z = \mathbf{b}^\top \mathbf{v}$, is also a Gaussian random variable, with mean $\mu = \mathbf{b}^\top \boldsymbol{\mu}$ and variance $\sigma^2 = \mathbf{b}^\top \boldsymbol{\Sigma} \mathbf{b}$.

We can exploit this result by noting that, if the variance ${\sigma_y}^2$ is known for $p(y|\mathbf{x})$, then $p(\mathbf{w}|\mathcal{D})$ is a multivariate Gaussian distribution. The random variable $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}$ is a linear weighting of the multivariate Gaussian RV $\mathbf{w}$. If $p(\mathbf{w}|\mathcal{D})$ is a $\mathcal{N}(\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n)$ then we know that

$$p(f(\mathbf{x})|\mathcal{D}) = \mathcal{N}(f(\mathbf{x})|\mu_x = \mathbf{x}\boldsymbol{\mu}_n, \sigma_x^2 = \mathbf{x}\boldsymbol{\Sigma}_n\mathbf{x}^\top)$$

The 95% credible interval for our prediction $f(\mathbf{x})$ is $[\mu_x - 1.96\sigma_x, \mu_x + 1.96\sigma_x]$.

In reality, we do not have ${\sigma_y}^2$. Consequently, the distribution $p(\mathbf{w}|\mathcal{D})$ is actually a Student-t distribution. An affine transformation of a multivariate Student-t does not have the same nice properties as the Gaussian. Instead, when we want to use the Bayesian linear regression model, it is typical to assume σ_y is the mode given by our NIG: ${\sigma_y}^2 = \frac{b_n}{a_n - 1}$. Given a specific σ_y, the distribution over $\mathbf{w}$ is Gaussian. This is a property of the NIG: if $p(\mathbf{w}, \sigma_y) = p(\mathbf{w}|\sigma_y)p(\sigma_y)$ is an NIG with parameters $\boldsymbol{\mu}, \boldsymbol{\Lambda}, a, n$ then $p(\mathbf{w}|\sigma_y)$ is Gaussian $\mathcal{N}(\boldsymbol{\mu}, {\sigma_y}^2\boldsymbol{\Lambda}^{-1})$.

The complete procedure is as follows.

1. Pick the hyperparameters $\lambda > 0$ for the prior over weights, and a_0 and b_0 for the prior over ${\sigma_y}^2$. They have less restrictive priors, you pick smaller values for λ (e.g., $\lambda = 0.01$) and can set $a_0 = b_0 = 1$.

2. Estimate the NIG $p(\mathbf{w}, {\sigma_y}^2|\mathcal{D})$, using formulas in Equation (15.2).

3. Get Gaussian posterior $p(\mathbf{w}|{\sigma_y}^2, \mathcal{D})$ by selecting ${\sigma_y}^2 = \frac{b_n}{a_n - 1}$ and computing $\boldsymbol{\Sigma}_n = {\sigma_y}^2\boldsymbol{\Lambda}^{-1}$ for $\boldsymbol{\Lambda} = \sum_{i=1}^n \mathbf{x}_i^\top \mathbf{x}_i + \lambda\mathbf{I}$ and $\boldsymbol{\mu}_n = \boldsymbol{\Lambda}^{-1}\sum_{i=1}^n \mathbf{x}_i^\top y_i$ to get $\mathcal{N}(\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n)$.

4. For any input $\mathbf{x}$, you can obtain $p(f(\mathbf{x})|\mathcal{D})$ as a Gaussian with $\mathcal{N}(\mathbf{x}\boldsymbol{\mu}_n, \mathbf{x}\boldsymbol{\Sigma}_n\mathbf{x}^\top)$.

We can constantly update the NIG posterior with new data, improving our estimates of $\mathbf{w}$ and of ${\sigma_y}^2$. For example, imagine we get m new samples, for a dataset of size $n+m$. We have already obtained $\boldsymbol{\mu}_n, \boldsymbol{\Lambda}_n, a_n, b_n$. We can update these with this new data by treating $\boldsymbol{\mu}_n, \boldsymbol{\Lambda}_n, a_n, b_n$ as the prior parameters, to get

$$\begin{aligned}
\boldsymbol{\Lambda}_{n+m} &= \sum_{i=n+1}^{n+m} \mathbf{x}_i^\top \mathbf{x}_i + \boldsymbol{\Lambda}_n \\
\boldsymbol{\mu}_{n+m} &= \boldsymbol{\Lambda}_{n+m}^{-1} \left(\sum_{i=n+1}^{n+m} \mathbf{x}_i^\top y_i + \boldsymbol{\Lambda}_n \boldsymbol{\mu}_n \right) \\
a_{n+m} &= a_n + \tfrac{1}{2} m \\
b_{n+m} &= b_n + \tfrac{1}{2} \left(\sum_{i=n+1}^{n+m} y_i^2 + \boldsymbol{\mu}_n^\top \boldsymbol{\Lambda}_n \boldsymbol{\mu}_n - \boldsymbol{\mu}_{n+m}^\top \boldsymbol{\Lambda}_{n+m} \boldsymbol{\mu}_{n+m} \right)
\end{aligned}$$

This is perfectly equivalent to having started with all of the $n+m$ samples and computing the posterior from $\boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0, a_0, b_0$. Bayesian linear regression, therefore, elegantly facilitates incorporating new data. Notice that this is one of the reasons we maintain $\boldsymbol{\Lambda}_n$ instead of $\boldsymbol{\Sigma}_n$, because we cannot easily update the inverted matrix with new data.

As we get more and more data, the variance of the NIG shrinks, as does the variance of $p(f(\mathbf{x})|\mathcal{D})$. Notice that the $\boldsymbol{\Sigma}_n$ shrinks because $\boldsymbol{\Lambda}_n = \sum_{i=1}^n \mathbf{x}_i^\top \mathbf{x}_i + \lambda \mathbf{I}$ grows with n. Consequently, $\mathbf{x}\boldsymbol{\Sigma}_n\mathbf{x}^\top$ also shrinks as we get more samples n. This indicates a reduction in uncertainty in our predictions as we get more data, in terms of identifying the best function in our linear function class. If f^* is in our linear function class, then eventually the mean converges to $f^*(\mathbf{x})$ and the interval shrinks to zero around this $f^*(\mathbf{x})$.

To reason about this a bit more formally, let us define

$$\mathbf{C}_n \doteq \frac{1}{n}(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}) \quad \text{where } \mathbf{X}^\top \mathbf{X} = \sum_{i=1}^n \mathbf{x}_i^\top \mathbf{x}_i.$$

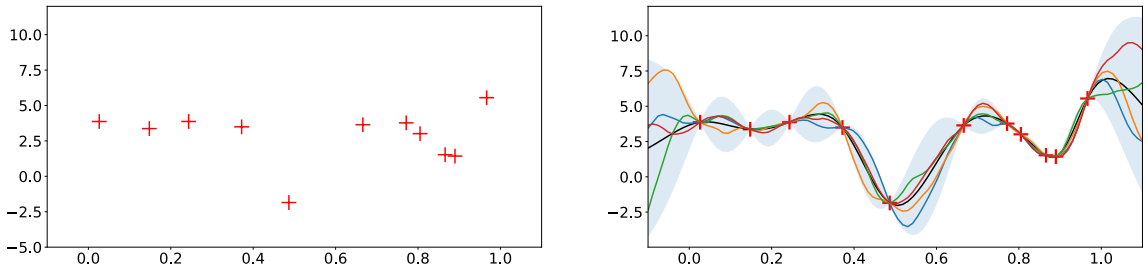
Notice that $\mathbf{C}_n \to \mathbf{C} \stackrel{\text{def}}{=} \mathbb{E}[\boldsymbol{X}\boldsymbol{X}^\top]$ as $n \to \infty$ (as we get more and more data), where $\boldsymbol{X}$ is the d-dimensional random variable for vector $\mathbf{x} \in \mathbb{R}^d$. Further, because we have $\lambda > 0$, we know that $\mathbf{C}_n$ is invertible for each n. Therefore, assuming that $\mathbf{C}$ is invertible, we know that $\mathbf{x}\mathbf{C}_n^{-1}\mathbf{x}^\top \to c_x$ as $n \to \infty$ for $c_x = \mathbf{x}\mathbf{C}^{-1}\mathbf{x}^\top$. We can write $\boldsymbol{\Sigma}_n = n^{-1}\mathbf{C}_n^{-1}$, giving

$$\mathbf{x}\boldsymbol{\Sigma}_n\mathbf{x}^\top = \mathbf{x}\left(n^{-1}\mathbf{C}_n^{-1}\right)\mathbf{x}^\top = n^{-1}\left(\mathbf{x}\mathbf{C}_n^{-1}\mathbf{x}^\top\right) \to 0.$$

Exercise 41: The above statement might need one more modifier: if f^* is in our linear function class, then eventually the mean converges to $f^*(\mathbf{x})$ as long as the prior does not put zero weight on the true weights $\mathbf{w}^*$. Why is this the case? And is it possible in Bayesian linear regression to pick a prior where $p(\mathbf{w}) = 0$ for some $\mathbf{w}$? □

15.3 The Nonlinear Setting & Gaussian Processes

Now we would like to get uncertainty estimates for nonlinear models. In other words, we assume that $Y = f^*(\mathbf{x}) + \epsilon$ where the true underlying function f^* can be a nonlinear function



(a) Data point observations **(b)** Five possible regression functions by GPR

Figure 15.1: The distribution over plausible functions, given by Gaussian process regression. The blue indicates the variability in the predictions given over this set of plausible functions, with each colored line corresponding to one possible function. There is in fact a continuum of possible functions, given by any lines within the blue region. This image is taken from the nice tutorial by Jie Wang [29].

and we still assume we have zero-mean Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ for some (unknown) $\sigma^2 > 0$. As before, the simplest way to extend to the nonlinear setting is to first map $\mathbf{x}$ to a new set of features $\boldsymbol{\phi}(\mathbf{x})$ and then do Bayesian linear regression on $\boldsymbol{\phi}(\mathbf{x})$. No further changes are required. We can simply compute $\boldsymbol{\Sigma}_n = {\sigma_y}^2 \boldsymbol{\Lambda}^{-1}$ using $\boldsymbol{\Lambda} = \sum_{i=1}^n \boldsymbol{\phi}_i^\top \boldsymbol{\phi}_i + \lambda \mathbf{I}$ and $\boldsymbol{\mu}_n = \boldsymbol{\Lambda}^{-1} \sum_{i=1}^n \boldsymbol{\phi}_i^\top y_i$ where we define $\boldsymbol{\phi}_i = \boldsymbol{\phi}(\mathbf{x}_i)$. Then we have that $p(f(\mathbf{x})|\mathcal{D})$ is $\mathcal{N}(\boldsymbol{\phi}(\mathbf{x})\boldsymbol{\mu}_n, \boldsymbol{\phi}(\mathbf{x})\boldsymbol{\Sigma}_n\boldsymbol{\phi}(\mathbf{x})^\top)$.

Exercise 42: Explicitly write the credible interval for the prediction $f(\mathbf{x})$, assuming we use these new features. □

Another direction is to use the *kernel trick* to extend to the nonlinear setting. We first explain the kernel trick and then how to use it to get Gaussian Processes by kernelizing Bayesian linear regression.

15.3.1 The Kernel Trick

In these notes we focus on the utility of kernels for representing functions. Our primary goal is to allow for larger hypothesis spaces, and to understand representability. Kernels, however, have also been popular in machine learning because of the property that they are inner products. This property has allowed for reformulations of certain optimization with a large number of features, with what is called the *kernel trick*.

Assume that you have features $\psi(\mathbf{x})$ and would like to do linear regression. We formulate the problem as

$$\sum_{i=1}^n (\psi(\mathbf{x}_i)\mathbf{w} - y_i)^2 = \|\boldsymbol{\Psi}\mathbf{w} - \mathbf{y}\|_2^2$$

where $\boldsymbol{\Psi}$ is composed of the vectors $\psi(\mathbf{x}_i)$, one on each row. Consider an alternative set of weights $\boldsymbol{\alpha} \in \mathbb{R}^n$ where we assume $\mathbf{w} = \boldsymbol{\Psi}^\top \boldsymbol{\alpha}$. Note that we can show that this does not constrain the solution, because our solution is projected to the space spanned by $\boldsymbol{\Psi}$; therefore we know that $\mathbf{w}$ is in the span of $\boldsymbol{\Psi}$, and so can be written as a linear combination

of its rows. It is equivalent to directly optimize for $\boldsymbol{\alpha}$, and our predictions will simply be

$$\begin{aligned}\psi(\mathbf{x})\mathbf{w} &= \psi(\mathbf{x})\boldsymbol{\Psi}^\top\boldsymbol{\alpha} \\ &= \sum_{i=1}^{n}\alpha_i\langle\psi(\mathbf{x}),\psi(\mathbf{x}_i)\rangle\end{aligned}$$

We can write this inner product as $k(\mathbf{x},\mathbf{x}_i) = \langle\psi(\mathbf{x}),\psi(\mathbf{x}_i)\rangle$. For certain ψ, this inner product has a simple closed-form solution, namely for the case where we have the kernel function. For example, if the ψ are polynomial features, this dot-product evaluates to the polynomial kernel. There are also underlying exponential features for the RBF kernel.[1] Therefore, if we can efficiently compute this dot product—as we can for kernels and their associated features spaces ψ—then it can actually be more efficient to instead solve the equivalent optimization

$$\|\boldsymbol{\Psi}\mathbf{w}-\mathbf{y}\|_2^2 = \|\boldsymbol{\Psi}\boldsymbol{\Psi}^\top\boldsymbol{\alpha}-\mathbf{y}\|_2^2 = \|\mathbf{K}\boldsymbol{\alpha}-\mathbf{y}\|_2^2$$

where $\mathbf{K}$ is the matrix composed of $k(\mathbf{x}_i,\mathbf{x}_j)$ at the entry (i,j): $K_{ij} = k(\mathbf{x}_i,\mathbf{x}_j)$.

15.3.2 Kernelizing Bayesian Linear Regression

We expand up to a large number of features $\boldsymbol{\phi}(\mathbf{x})$, and then reformulate so that we only ever have $\boldsymbol{\phi}(\mathbf{x})\boldsymbol{\phi}(\mathbf{x}')^\top$. More precisely, we include the the prior parameter $\boldsymbol{\Lambda}_0$ into this dot product to get a weighted dot product $\boldsymbol{\phi}(\mathbf{x})\boldsymbol{\Lambda}_0\boldsymbol{\phi}(\mathbf{x}')^\top$. This weighted dot product corresponds to the kernel function $k(\mathbf{x},\mathbf{x}')$. This is precisely what is done in Gaussian Process Regression. We can visualize a GP in Figure 15.1, where there is a shaded region of plausible function predictions, based on the dataset given by the red crosses.

The choice of $\boldsymbol{\Lambda}_0$ defines the width of the activation region for the kernel. For example, in the standard radial basis function kernel, we typically use $\boldsymbol{\Lambda}_0 = \lambda\mathbf{I}$ and have $\exp(-\frac{\lambda}{2}\|\mathbf{x}-\mathbf{x}'\|_2^2)$. We can think of λ^{-1} as the variance or the width. Smaller λ result in a wider activation, so that $\mathbf{x}'$ further from $\mathbf{x}$ still have non-negligible values $k(\mathbf{x},\mathbf{x}')$. Bigger λ result in a small width, and so only very nearby $\mathbf{x}'$ have non-negligible activation. The choice of λ for GPs ends up corresponding to the choice of width in the kernel, and is a hyperparameter that needs to be tuned.

Now let us see how GPs maintain $p(f(\mathbf{x})|\mathcal{D})$. We can start by examining the formula and then discuss where it comes from. Let the dataset be composed of matrix $\mathbf{X}\in\mathbb{R}^{n\times d}$ and targets $\mathbf{y}\in\mathbb{R}^n$. For any $\mathbf{x}$, which need not be in the training set, let $k(\mathbf{x},\mathbf{X}) = [k(\mathbf{x}_1,\mathbf{x}),k(\mathbf{x}_2,\mathbf{x}),\ldots,k(\mathbf{x}_n,\mathbf{x})]$ be the vector of kernel (similarity) values between $\mathbf{x}$ and each training datapoint $\mathbf{x}_i$. Let $\mathbf{K} = k(\mathbf{X},\mathbf{X})\in\mathbb{R}^{n\times n}$ be the kernel matrix for our training dataset, where $K_{ij} = k(\mathbf{x}_i,\mathbf{x}_j)$. Then we have that

$$p(f(\mathbf{x})|\mathcal{D}) = \mathcal{N}\left(k(\mathbf{x},\mathbf{X})(\mathbf{K}+\sigma^2\mathbf{I})^{-1}\mathbf{y},\ k(\mathbf{x},\mathbf{x})-k(\mathbf{x},\mathbf{X})(\mathbf{K}+\sigma^2\mathbf{I})^{-1}k(\mathbf{x},\mathbf{X})^\top\right) \quad (15.3)$$

[1] In fact, the ψ for the RBF kernel is actually infinite-dimensional. This is one of the motivations for the kernel trick: the setting where the number of features is very large or even infinite. Such a large set of features should be beneficial for modeling, but is expensive to use. The kernel trick allows us to implicitly use such features, without paying the computational cost. Of course, we swap the cost of computing these features with the cost of computing the kernel matrix for the whole dataset, which itself can be expensive. As usual, nothing is free.

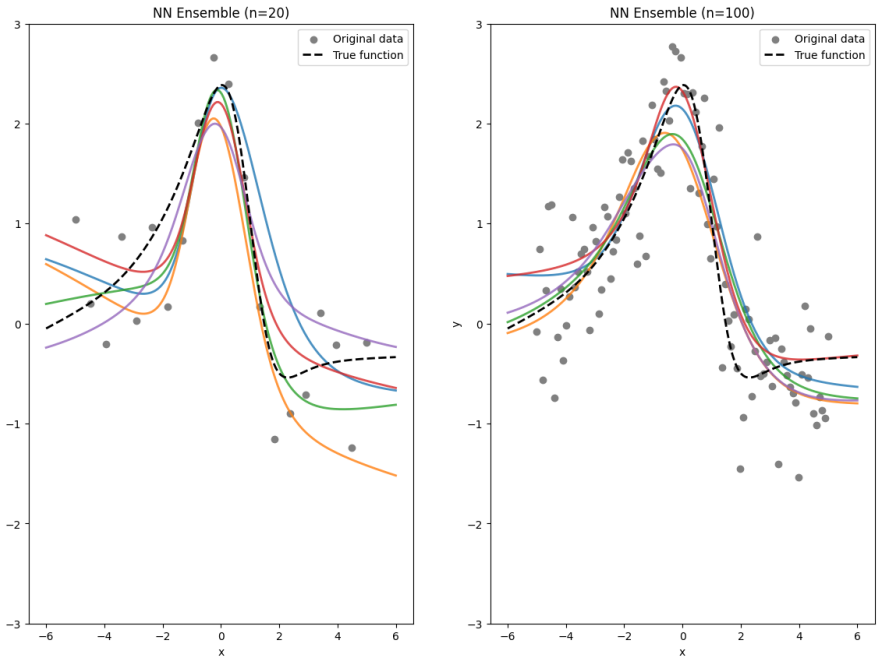


Figure 15.2: An ensemble of 5 for a datasets of size 20 (left-hand side plot) and a dataset of size 100 (right-hand side plot). The true data is generated from a neural network in the same function class as the neural networks that are learned on the data.

This formula assumes we have σ^2. Again, this is a hyperparameter that needs to be tuned for the GP.

We can reason about why this formula makes sense by first considering the mean component. Notice that $\mathbf{w} = (\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{y}$ corresponds to the solution from kernel regression, where σ^2 is the regularization parameter for ℓ_2 regularization. The term $\boldsymbol{\phi}(\mathbf{x}) = k(\mathbf{x}, \mathbf{X})$ is the new kernel features. Therefore, $k(\mathbf{x}, \mathbf{X}, \mathbf{x})(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{y} = \boldsymbol{\phi}(\mathbf{x})\mathbf{w}$, which is precisely the prediction given when we used the kernel trick in regression.

Obtaining the covariance term is a bit more complex. Conceptually, it is simple: we simply need to do some algebra rearranging terms so that we always have $\boldsymbol{\phi}(\mathbf{x})$ as a dot product with another $\boldsymbol{\phi}(\mathbf{x}')$. We do not go through these steps here; for these steps see Equation 2.12 in [22].

15.4 Uncertainty Estimation for Neural Networks using Ensembles

We use a different approach for uncertainty estimation with neural networks because obtaining the posterior over parameters is much more difficult. If we could compute or easily sample from $p(f_{\mathbf{w}}(\mathbf{x})|\mathcal{D})$ for our NN, then we could get intervals around our predictions $f_{\mathbf{w}}(\mathbf{x})$. However, it is computationally expensive to samples from this posterior, $p(f_{\mathbf{w}}(\mathbf{x})|\mathcal{D})$. Research around this is active, in a topic called *Bayesian Neural Networks*, but is insufficiently mature to cover here.

Instead, we turn to ensembles, which is a generic idea that can be layered on top of any

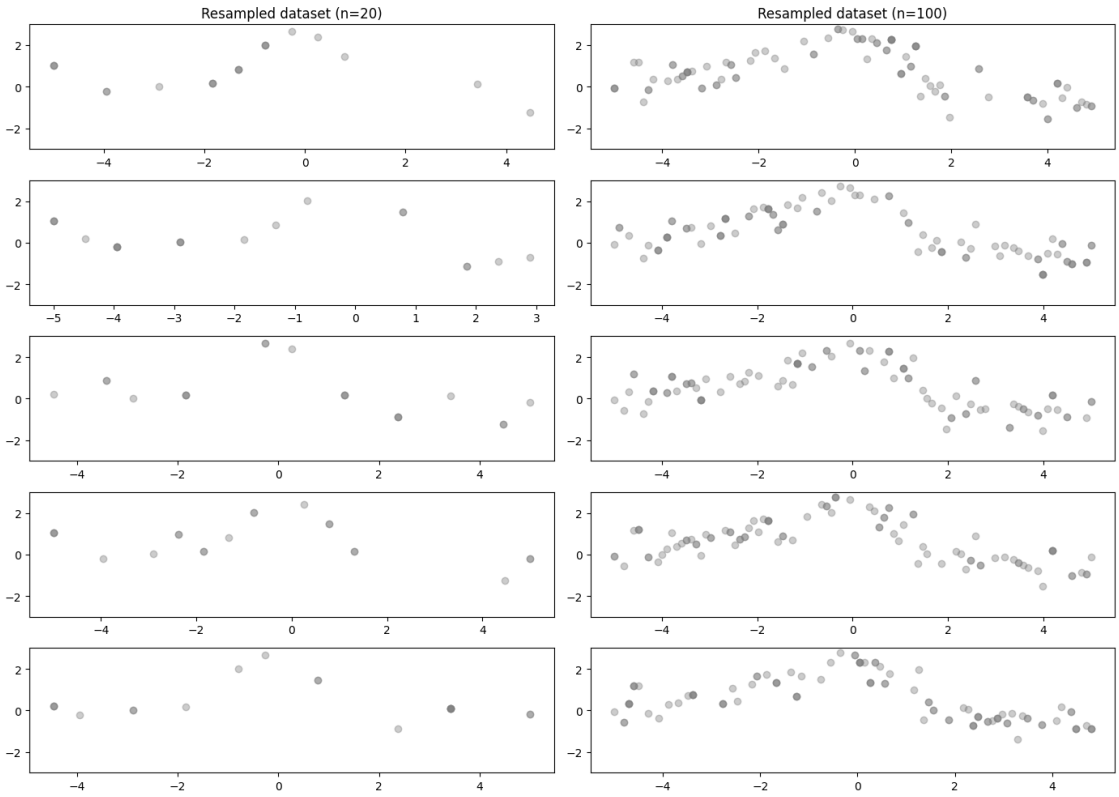


Figure 15.3: The resampled datasets used to train the functions in Figure 15.2.

learning algorithm. The idea is to simulate different weights that we could have learned, if we had seen different data. Recall that our uncertainty stems from the fact that we have only one (limited) dataset. Potentially, if we had seen a different dataset, we would make a different prediction for $\mathbf{x}$. In an ideal scenario, we could get m different datasets of size n, estimate $\mathbf{w}_1, \ldots, \mathbf{w}_m$ from those datasets $\mathcal{D}_1, \ldots, \mathcal{D}_m$ and observe the spread in their predictions. For example, for a given point $\mathbf{x}$, we would query the m neural networks $\hat{y}_1 \doteq f_{\mathbf{w}_1}(\mathbf{x}), \ldots, \hat{y}_m \doteq f_{\mathbf{w}_m}(\mathbf{x})$ and could observe the spread in predictions. We could compute a mean $\mu = \frac{1}{m}\sum_{k=1}^{m} \hat{y}_k$ and 95% confidence interval v over $\hat{y}_1, \ldots, \hat{y}_m$ to get our uncertainty $[\mu - v, \mu + v]$. But, we do not get m datasets, so what do we do?

The idea behind ensembles is to treat your one dataset $\mathcal{D}$ as an empirical distribution and resample from it. This is called bootstrap resampling in statistics. We sample n from $\mathcal{D}$, with replacement, to create $\mathcal{D}_1$, and repeat this process m times to get our m datasets. Then we train m neural networks independently on each dataset. We could also have taken this approach for the linear setting, instead of using Bayesian linear regression, and can use it to get uncertainty estimates for our other generalized linear models (GLMs). With neural networks, the diversity of the ensemble is both affected by the dataset and the random initializations to the networks, which might result in different local minima. With linear models, the diversity of the ensemble is strictly from seeing different datasets.[2] The use

[2]Note that for neural networks it has been observed that the random initialization already provides sufficient diversity, and it is not necessary (or even at times harmful) to resample different datasets [**?**]. This

of ensembles for uncertainty estimation is depicted in Figure 15.2, with the corresponding resampled datasets in Figure 15.3.

The primary benefits of using ensembles is the simplicity of the approach, but there are some downsides. One is that a large ensemble may be needed to get effective uncertainty estimates, which can be expensive. Training one neural network can be expensive, let alone m of them. Another issue is that the ensemble can often be an underestimate of uncertainty, primarily if our dataset $\mathcal{D}$ is small.

approach is unprincipled, and so we do not advocate for it here.

Chapter 16

Learning on Temporal Data

Most real world problems have data with temporal dependencies. We see a sequence of observations $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$, such as the weather information for a city measured every 10 minutes or camera snapshots from a self-driving car. For the self-driving car, we may want to predict the presence and locations of objects in the image (e.g., pedestrians) within the next 10 seconds. For weather, we may want to predict (forecast) the weather in the next 10 minutes, next hour and next day, multiple steps into the future.

In this setting, we no longer have i.i.d. data, and need to consider new learning strategies. Our goals will still be to learn accurate (nonlinear) predictors or generative models, potentially in the presence of missing data. But now we will need to condition on histories of observations, to make accurate predictions, as we will discuss in Section 16.1. Further, when using mini-batch SGD updating, we need to ensure we preserve the temporal structure when shuffling our updates.

This problem setting might seem more complicated, but actually in some ways is beneficial. Once we have temporal data, we can actually overcome some of the partial observability arising from our limited measurements and sensing. Consider for example having one snapshot of a frisbee flying from your window. It would be difficult to make an accurate prediction of which direction it is going or its speed. With a sequence of snapshots, it is easy to predict its direction, its speed and where the frisbee will be next. Sequences give us more context to make predictions, and in this chapter you will see algorithms that can use such sequences.

16.1 Conditioning on History

Assume we have a sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$. For example, $\mathbf{x}_i$ could be a vector describing the weather in Edmonton, consisting of the current temperature, humidity, precipitation amount, and wind speed. These $\mathbf{x}_i$ could be measured every ten minutes, with $\mathbf{x}_1$ the information on July 1, 9 am, $\mathbf{x}_2$ the information on July 1, 9:10 am, and so on. Once we observe $\mathbf{x}_t$, we may want to predict $\mathbf{x}_{t+1}$ (all four pieces of weather info in 10 minutes), or $\mathbf{x}_{t+10}$ (all four pieces of weather info in 100 minutes) or just the temperature at some point(s) in the future. We can set $\mathbf{y}_t$ to be any of these targets, given context $\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1$.

There are two common scenarios: using a fixed-length window for the context and using the whole history. For the first, we effectively turn our temporal problem into a standard supervised problem. Assume we pick a context length back in time of length p. We create

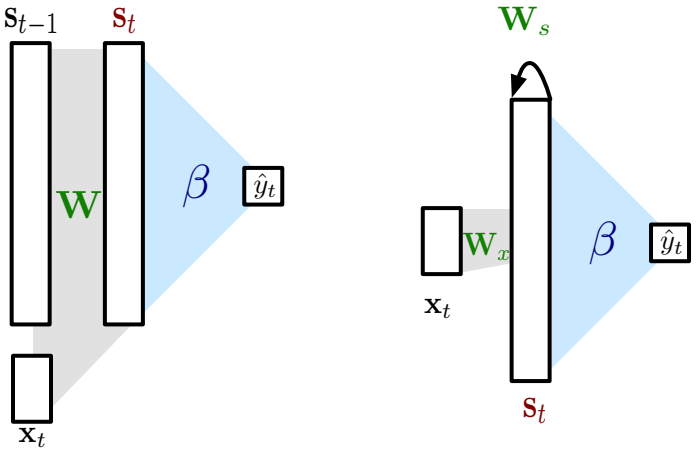


Figure 16.1: Two different ways to view a simple RNN, with $\mathbf{s}_{t+1} = ReLU(\mathbf{s}_t\mathbf{W}_s + \mathbf{x}_{t+1}\mathbf{W}_x)$ *and* $\hat{y}_t = \mathbf{s}_t\boldsymbol{\beta}$*. The first image unrolls one step, showing the previous state and the most recent observation being input into the neural network. The second shows this recurrent relationship with a self-arrow.*

a new dataset consisting of samples

$$
\begin{aligned}
&(\tilde{\mathbf{x}}_1 = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_p], \tilde{\mathbf{y}}_1 = \mathbf{y}_p) \\
&(\tilde{\mathbf{x}}_2 = [\mathbf{x}_2, \mathbf{x}_2, \ldots, \mathbf{x}_{p+1}], \tilde{\mathbf{y}}_2 = \mathbf{y}_{p+1}) \\
&\ldots \\
&(\tilde{\mathbf{x}}_{n-p+1} = [\mathbf{x}_{n-p+1}, \mathbf{x}_{n-p+2}, \ldots, \mathbf{x}_n], \tilde{\mathbf{y}}_{n-p+1} = \mathbf{y}_n)
\end{aligned}
$$

We then use this new dataset with any learning algorithm, for example we can train a multilayer neural network.

The key issue with this approach is that we are limited to fixed length histories p. But, dependencies might go further back-in-time. For example, if we are generating the next word for a paragraph, we may want to have dependencies back to the very beginning of the paragraph or even document. We would have to pick a very large p. Further, in some cases, we need to look further back-in-time, and for others recent information is enough. With a fixed p, we likely need to pick a very large p to cover both cases. Instead, we want to be able to handle variable-length dependencies back-in-time, which is exactly what recurrent neural networks do, described in the next section.

16.2 Recurrent Neural Networks

A recurrent neural network (RNN) builds a fixed-size, compact summary $\mathbf{s}$ of the history. It does so by updating $\mathbf{s}$ recursively

$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{x}_{t+1})$$

We sometimes call f the state-update function, because the $\mathbf{s}_t$ can be seen as a hidden state that the RNN is inferring from the history of observations $\mathbf{x}_i$. A typical, simple variant of

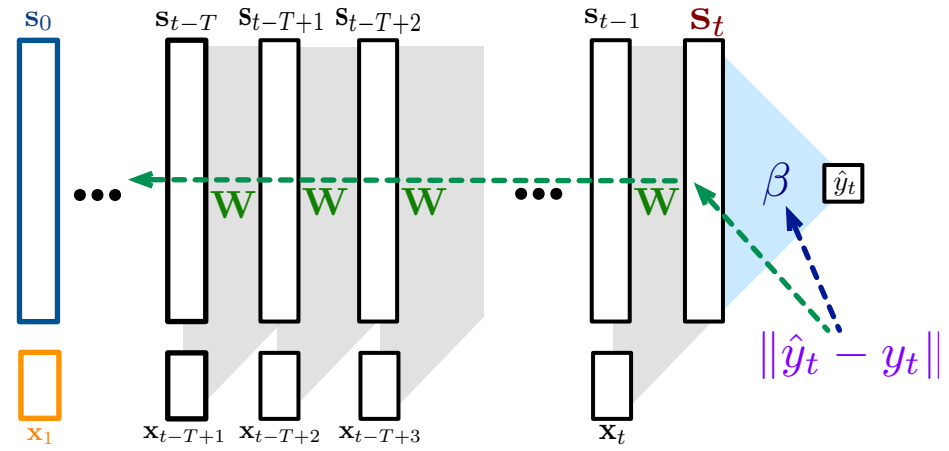


Figure 16.2: Backpropagation through time (BPTT) for getting the gradient for an RNN.

this state-update function is to use a linear transformation and some nonlinear activation, like a ReLU

$$\mathbf{s}_{t+1} = \text{ReLU}(\mathbf{s}_t \mathbf{W}_s + \mathbf{x}_{t+1} \mathbf{W}_x)$$

Just as before, though, much more complex neural network architectures can be used. This $\mathbf{s}_t$ corresponds to a hidden layer in the RNN, as shown in Figure 16.1.

Because we maintain the same differentiability properties, it is reasonably straightforward to compute the gradient. The recurrence, however, causes dependence all the way back in-time. This is made clear by unrolling the RNN. To see why, assume we get the prediction for our targets $\mathbf{y}_t$ using the current state $\mathbf{s}_t$ created by the RNN, with a learned NN $\hat{\mathbf{y}}_t \doteq g_\beta(\mathbf{s}_t)$. If we want to compute the gradient for time t, with target $\mathbf{y}_t$, we have error

$$\begin{aligned}
\|\hat{\mathbf{y}}_t - \mathbf{y}_t\|_2^2 &= \|g_\beta(\mathbf{s}_t) - \mathbf{y}_t\|_2^2 \\
&= \|g_\beta(f_W(\mathbf{s}_{t-1}, \mathbf{x}_t) - \mathbf{y}_t\|_2^2 \\
&= \|g_\beta(f_W(f_W(\mathbf{s}_{t-2}, \mathbf{x}_{t-1}), \mathbf{x}_t) - \mathbf{y}_t\|_2^2 \\
&\dots \\
&= \|g_\beta(f_W(f_W(\dots(f_W(\mathbf{s}_0, \mathbf{x}_1)\dots), \mathbf{x}_{t-1}), \mathbf{x}_t) - \mathbf{y}_t\|_2^2
\end{aligned}$$

The gradient with respect to $\mathbf{W}$ for this loss has to consider how small changes to $\mathbf{W}$ would have effected all the states $\mathbf{s}_1, \dots, \mathbf{s}_t$ created until this point. This is called *backpropagation through time* (BPTT), depicted in Figure 16.2.

16.3 Transformers

Read these very clear notes by John Thickstun [27]. More details for this section in the future.

Bibliography

[1] A Banerjee, S Merugu, I S Dhillon, and J Ghosh. Clustering with bregman divergences. *Journal of Machine Learning Research*, 2005.

[2] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[3] Peter L Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 2002.

[4] Stephen Bates, Trevor Hastie, and Robert Tibshirani. Cross-validation: What does it estimate and how well does it do it? *arXiv:2104.00673*, 2021.

[5] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Imaging Sciences*, 2009.

[6] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias?variance trade-off. *Proceedings of the National Academy of Sciences of the United States of America*, 2019.

[7] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When Is Nearest Neighbor Meaningful? In *Database Theory*, volume 1540. Springer Berlin Heidelberg, 1999.

[8] Olivier Bousquet, Stephane Boucheron, and Gabor Lugosi. Introduction to Statistical Learning Theory. In *Advanced Lectures on Machine Learning*. Springer Berlin Heidelberg, 2004.

[9] Gavin C Cawley and Nicola LC Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *The Journal of Machine Learning Research*, 2010.

[10] Carl Doersch. Tutorial on Variational Autoencoders. *arXiv:1606.05908 [cs, stat]*, 2021.

[11] Torsten Hoe?er, Dan Alistarh, Tal Ben-Nun, and Nikoli Dryden. Sparsity in Deep Learning: Pruning and growth for e?cient inference and training in neural networks. *The Journal of Machine Learning Research*, 2021.

[12] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer New York, 2013.

[13] Sham M Kakade, Karthik Sridharan, and Ambuj Tewari. On the complexity of linear prediction: Risk bounds, margin bounds, and regularization. In *Advances in Neural Information Processing Systems*, 2008.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[15] Frederik Kunstner, Raunak Kumar, and Mark Schmidt. Homeomorphic-Invariance of EM: Non-Asymptotic Convergence in KL Divergence for Exponential Families via Mirror Descent. In *International Conference on AI and Statistics*, 2021.

[16] Ilja Kuzborskij, Csaba Szepesvári, Omar Rivasplata, Amal Rannen-Triki, and Razvan Pascanu. On the Role of Optimization in Double Descent: A Least Squares Study. In *Advances in Neural Information Processing Systems*, 2021.

[17] Jorge Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 1980.

[18] Bruno A Olshausen and David J Field. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research*, 1997.

[19] Neal Parikh and Stephen Boyd. Proximal Algorithms. *Foundations and Trends in Optimization*, 1(3), 2014.

[20] J. Park and I. W. Sandberg. Universal Approximation Using Radial-Basis-Function Networks. *Neural Computation*, 3(2), 1991.

[21] K B Petersen. The matrix cookbook. *Technical University of Denmark*, 2004.

[22] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, Mass, 2006.

[23] Matthew Schlegel, Yangchen Pan, Jiecao Chen, and Martha White. Adapting Kernel Representations Online Using Submodular Maximization. In *International Conference on Machine Learning*, 2017.

[24] Robin M. Schmidt, Frank Schneider, and Philipp Hennig. Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers, 2021.

[25] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research*, 2019.

[26] Samuel L. Smith, Benoit Dherin, David G. T. Barrett, and Soham De. On the Origin of Implicit Regularization in Stochastic Gradient Descent. In *International Conference on Learning Representations*, 2021.

[27] John Thickstun. The transformer model in equations. In *Course Notes*, 2024.

[28] Gitte Vanwinckelen and Hendrik Blockeel. On estimating model accuracy with repeated cross-validation. In *BeneLearn 2012: Proceedings of the 21st Belgian-Dutch Conference on Machine Learning*, 2012.

[29] Jie Wang. An Intuitive Tutorial to Gaussian Processes Regression. *arXiv:2009.10862 [cs, stat]*, 2021.

[30] Martha White. *Regularized Factor Models*. PhD thesis, University of Alberta, 2014.

[31] Martha White. *Basics of Machine Learning*. 2020.

[32] Matthew D Zeiler. ADADELTA: An adaptive learning rate method. *arXiv.org*, 2012.

Appendix A

Extra Information

You will not be tested on anything in the appendix. It is simply here for your interest.

A.1 More on Linear Regression

The following section is an expanded version of Section 3.2.3, with derivation steps included.

A.1.1 The Bias-Variance Trade-off

A natural question to ask is how this regularization parameter can be selected, and the impact on the final solution vector. The selection of this regularization parameter leads to a bias-variance trade-off. To understand this trade-off, we need to understand what it means for the solution to be biased, and how to characterize the variance of the solution, across possible datasets.

Let us begin by presuming that the distributional assumptions behind linear regression are true. This means that there exists a true parameter $\boldsymbol{\omega}$ such that for each of the data points $Y_i = \sum_{j=0}^{d} \omega_j X_{ij} + \varepsilon_i$, where the ε_j are i.i.d. random variables drawn according to $\mathcal{N}(0, \sigma^2)$. We can characterize the solution vector (estimator) $\mathbf{w}_{\text{MLE}}$ as a random variable, where the randomness is across possible datasets that could have been observed. In this sense, we are considering the dataset $\mathcal{D}$ to be a random variable, and the solution $w_{\text{MLE}}(\mathcal{D})$ from that dataset as a function of this random variable.

The reason we care about the bias and variance of $\mathbf{w}_{\text{MLE}}$ because the expected mean-squared error to the true weights can be decomposed into the bias and variance.

$$\mathbb{E}\left[\|\mathbf{w}(\mathcal{D}) - \boldsymbol{\omega}\|_2^2\right] = \mathbb{E}\left[\sum_{j=1}^{d} (w_j(\mathcal{D}) - \omega_j)^2\right] = \sum_{j=1}^{d} \mathbb{E}\left[(w_j(\mathcal{D}) - \omega_j)^2\right]$$

where we can then further simplify this inner term

$$\begin{aligned}\mathbb{E}\left[(w_j(\mathcal{D}) - \omega_j)^2\right] &= \mathbb{E}\left[(w_j(\mathcal{D}) - \mathbb{E}\left[w_j(\mathcal{D})\right] + \mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j)^2\right] \\ &= \mathbb{E}\left[(w_j(\mathcal{D}) - \mathbb{E}\left[w_j(\mathcal{D})\right])^2\right] + \mathbb{E}\left[(\mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j)^2\right]\end{aligned}$$

where the second step follows from the fact that

$$\begin{aligned}-2\mathbb{E}\left[(w_j(\mathcal{D}) - \mathbb{E}\left[w_j(\mathcal{D})\right])(\mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j)\right] &= (\mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j)\mathbb{E}\left[w_j(\mathcal{D}) - \mathbb{E}\left[w_j(\mathcal{D})\right]\right] \\ &= 0.\end{aligned}$$

The first term above in $\mathbb{E}\left[(w_j(\mathcal{D}) - \mathbb{E}\left[w_j(\mathcal{D})\right])^2\right]$ is the variance of the jth weight and the second term is the bias of the jth weight, where $\mathbb{E}\left[(\mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j)^2\right] = (\mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j)^2$ because nothing is random in this term so the outer expectation is dropped. This gives

$$\begin{aligned}\mathbb{E}\left[\|\mathbf{w}(\mathcal{D}) - \boldsymbol{\omega}\|_2^2\right] &= \sum_{j=1}^{d} \mathbb{E}\left[(w_j(\mathcal{D}) - \omega_j)^2\right] \\ &= \sum_{j=1}^{d} (\mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j)^2 + \mathrm{Var}\left[w_j(\mathcal{D})\right] \end{aligned} \tag{A.1}$$

showing that the expected mean-squared error to the true weight vector $\boldsymbol{\omega}$ decomposes into the squared bias—where the bias is $\mathbb{E}\left[w_j(\mathcal{D})\right] - \omega_j$—and the variance.

The bias-variance trade-off reflects the fact that we could potentially reduce the mean-squared error by incurring some bias, as long as the variance is decreased more than the squared bias. Note that we do not directly optimize the bias-variance trade-off. We cannot actually measure the bias, so we do not directly minimize these terms. Rather, this decomposition guides how we select model classes.

Let us now look at the expected value (with respect to training data set $\mathcal{D}$) for the weight vector w_{MLE}, with $\boldsymbol{\varepsilon} = (\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n)$:

$$\begin{aligned}\mathbb{E}[w_{\mathrm{MLE}}(\mathcal{D})] &= \mathbb{E}\left[\left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top\left(\mathbf{X}\boldsymbol{\omega} + \boldsymbol{\varepsilon}\right)\right] \\ &= \mathbb{E}\left[\left(\mathbf{X}^\top\mathbf{X}\right)^{-1}(\mathbf{X}^\top\mathbf{X})\boldsymbol{\omega}\right] + \mathbb{E}\left[\left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top\boldsymbol{\varepsilon}\right] \\ &= \mathbb{E}\left[\boldsymbol{\omega}\right] + \mathbb{E}\left[\left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top\right]\mathbb{E}\left[\boldsymbol{\varepsilon}\right] \\ &= \boldsymbol{\omega},\end{aligned}$$

where the third equality follows from the fact that the noise terms $\boldsymbol{\varepsilon}$ are independent of the features and the last equality because ω is a constant vector (non-random) and $\mathbb{E}[\boldsymbol{\varepsilon}] = \mathbf{0}$. An estimator whose expected value is the true value of the parameter is called an *unbiased estimator*.

The covariance matrix for the optimal set of parameters can be expressed as

$$\begin{aligned}\mathrm{Cov}[w_{\mathrm{MLE}}(\mathcal{D})] &= \mathbb{E}\left[\left(w_{\mathrm{MLE}}(\mathcal{D}) - \boldsymbol{\omega}\right)\left(w_{\mathrm{MLE}}(\mathcal{D}) - \boldsymbol{\omega}\right)^\top\right] \\ &= \mathbb{E}\left[w_{\mathrm{MLE}}(\mathcal{D})w_{\mathrm{MLE}}(\mathcal{D})^\top\right] - \boldsymbol{\omega}\boldsymbol{\omega}^\top\end{aligned}$$

Taking[1] $\mathbf{X}^\dagger = \left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top$, we have $w_{\mathrm{MLE}}(\mathcal{D}) = \boldsymbol{\omega} + \mathbf{X}^\dagger\boldsymbol{\varepsilon}$, so

$$\begin{aligned}\mathrm{Cov}[w_{\mathrm{MLE}}(\mathcal{D})] &= \mathbb{E}\left[\left(\boldsymbol{\omega} + \mathbf{X}^\dagger\boldsymbol{\varepsilon}\right)\left(\boldsymbol{\omega} + \mathbf{X}^\dagger\boldsymbol{\varepsilon}\right)^\top\right] - \boldsymbol{\omega}\boldsymbol{\omega}^\top \\ &= \boldsymbol{\omega}\boldsymbol{\omega}^\top + \mathbb{E}\left[\mathbf{X}^\dagger\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top\mathbf{X}^{\dagger\top}\right] - \boldsymbol{\omega}\boldsymbol{\omega}^\top\end{aligned}$$

[1] This matrix is called the pseudo-inverse of $\mathbf{X}$. The idea of a pseudo-inverse generalizes the concept of inverses to non-invertible matrices, including rectangular matrices. This includes low-rank $\mathbf{X}$, where the pseudoinverse uses the inverse of non-zero singular values and otherwise sets the entry to zero. It is a useful concept, but not one we will need to use again and so is not explained in-depth here.

because $\mathbb{E}\left[\mathbf{X}^\dagger \boldsymbol{\varepsilon}\boldsymbol{\omega}^\top\right] = \mathbb{E}\left[\mathbf{X}^\dagger\right]\mathbb{E}\left[\boldsymbol{\varepsilon}\right]\boldsymbol{\omega}^\top = \mathbf{0}$. Now because the noise terms are independent of the inputs, i.e., $\mathbb{E}\left[\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top|\mathbf{X}\right] = \mathbb{E}\left[\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top\right] = \sigma^2\mathbf{I}$, we can use the law of total probability (also called the tower rule), to get

$$\begin{aligned}\mathbb{E}\left[\mathbf{X}^\dagger\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top\mathbf{X}^{\dagger\top}\right] &= \mathbb{E}\left[\mathbb{E}\left[\mathbf{X}^\dagger\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top\mathbf{X}^{\dagger\top}|\mathbf{X}\right]\right] \\ &= \mathbb{E}\left[\mathbf{X}^\dagger\mathbb{E}\left[\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top|\mathbf{X}\right]\mathbf{X}^{\dagger\top}\right] \\ &= \sigma^2\mathbb{E}\left[\mathbf{X}^\dagger\mathbf{X}^{\dagger\top}\right].\end{aligned}$$

Thus, we have

$$\mathrm{Cov}[w_{\mathrm{MLE}}(\mathcal{D})] = \sigma^2\mathbb{E}\left[(\mathbf{X}^\top\mathbf{X})^{-1}\right] = \sigma^2\mathbb{E}\left[\mathbf{V}\boldsymbol{\Sigma}_d^{-2}\mathbf{V}^\top\right]$$

where σ^2 is the variance of y given x. Naturally, the covariance of the weights across datasets is higher if the variance of the targets is higher.

As discussed above, the matrix $\mathbf{X}^\top\mathbf{X} = \mathbf{V}\boldsymbol{\Sigma}\mathbf{V}^\top$ can be poorly conditioned, with some zero or near-zero singular values. Consequently, this covariance matrix can be poorly conditioned, with high magnitude co-variance values. This implies that, across datasets, the solution $w_{\mathrm{MLE}}(\mathcal{D})$ can vary widely. This type of behavior is suggestive of overfitting, and is not desirable. If our solution could be very different across several different random subsets of data, we cannot be confident in any one of these solutions.

We can also reason about how this variance suggests high MSE. To characterize the MSE, we only need the diagonal of this covariance matrix, as shown in Equation (A.1). We need $\sum_{j=1}^d \mathrm{Var}\left[w_j(\mathcal{D})\right]$, which is the sum of the diagonals of this covariance matrix, also called the *trace* of the matrix. Fortunately for us, the trace of a square matrix corresponds to the sum of the eigenvalues of the matrix, which for the above are $\boldsymbol{\Sigma}_d^{-2}$. Therefore we have that

$$\begin{aligned}\sum_{j=1}^d \mathrm{Var}\left[w_{\mathrm{MLE},j}(\mathcal{D})\right] &= \mathrm{trace}(\mathrm{Cov}[w_{\mathrm{MLE}}(\mathcal{D})]) \\ &= \sigma^2\mathbb{E}\left[\mathrm{trace}\left(\mathbf{V}\boldsymbol{\Sigma}_d^{-2}\mathbf{V}^\top\right)\right] \qquad \triangleright \text{ linearity of trace} \\ &= \sigma^2\mathbb{E}\left[\sum_{j=1}^d \sigma_j^{-2}\right].\end{aligned}$$

This formula makes it clear that the variance of the weights is tied to the magnitudes of the inverse of the singular values. If we have very small singular values, then this sum is much larger. The singular values will not be small for all datasets that could have been observed, but in cases where overfitting is possible (small n), we expect it to happen for a large proportion of datasets.

The regularized solution, on the other hand, is much less likely to have high covariance, but will no longer be unbiased. Let $w_{\mathrm{MAP}}(\mathcal{D})$ be the MAP estimate for the ℓ_2 regularized problem with $\lambda > 0$. Using a similar analysis to above, the expected value of $w_{\mathrm{MAP}}(\mathcal{D})$ is

$$\mathbb{E}[w_{\mathrm{MAP}}(\mathcal{D})] = \mathbb{E}\left[\left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^\top\left(\mathbf{X}\boldsymbol{\omega} + \boldsymbol{\varepsilon}\right)\right] = \mathbb{E}\left[\left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\left(\mathbf{X}^\top\mathbf{X}\right)\boldsymbol{\omega}\right] \neq \boldsymbol{\omega}.$$

As $\lambda \to 0$, the MAP solution gets closer and closer to being unbiased. If we let $\mathbf{\Lambda}_d$ be a diagonal matrix with values $\sigma_j^2/(\sigma_j^2+\lambda)^2$ on the diagonal, then the covariance is

$$\begin{aligned}\text{Cov}[w_{\text{MAP}}(\mathcal{D})] &= \sigma^2\mathbb{E}\left[(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I})^{-1}(\mathbf{X}^\top\mathbf{X})(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I})^{-1}\right] \\ &= \sigma^2\mathbb{E}\left[\mathbf{V}\mathbf{\Lambda}_d\mathbf{V}^\top\right].\end{aligned}$$

because $\mathbf{X}^\top\mathbf{X} = \mathbf{V}\Sigma_d^2\mathbf{V}^\top$ and $\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I} = \mathbf{V}\Sigma_d^2\mathbf{V}^\top + \lambda\mathbf{I} = \mathbf{V}(\Sigma_d^2+\lambda\mathbf{I})\mathbf{V}^\top$ giving

$$\begin{aligned}(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I})^{-1}(\mathbf{X}^\top\mathbf{X})(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I})^{-1} &= \mathbf{V}(\Sigma_d^2+\lambda\mathbf{I})^{-1}\mathbf{V}^\top\mathbf{V}\Sigma_d^2\mathbf{V}^\top\mathbf{V}(\Sigma_d^2+\lambda\mathbf{I})^{-1}\mathbf{V}^\top \\ &= \mathbf{V}(\Sigma_d^2+\lambda\mathbf{I})^{-1}\Sigma_d^2(\Sigma_d^2+\lambda\mathbf{I})^{-1}\mathbf{V}^\top \qquad \triangleright\ \mathbf{V}^\top\mathbf{V}=\mathbf{I} \\ &= \mathbf{V}\mathbf{\Lambda}_d\mathbf{V}^\top \qquad \triangleright\ \mathbf{\Lambda}_d \doteq \Sigma_d^2(\Sigma_d^2+\lambda\mathbf{I})^{-2}.\end{aligned}$$

This covariance is much less susceptible to ill-conditioned $\mathbf{X}^\top\mathbf{X}$, because the shift by λ improves the condition. The covariance is now dictated instead by the eigenvalues $\mathbf{\Lambda}_d$, which only have $\sigma_d^2+\lambda$ on the denominator. Consequently, we expect w_{MAP} to have lower variance across different datasets. This correspondingly implies that we are less likely to overfit to any one dataset. Notice that as $\lambda \to \infty$, the variance decreases to zero, but the bias increases to its maximal value (i.e., the norm of the true weights). There is an optimal choice of λ that minimizes this bias-variance trade-off—if we could find it.

A.2 More on Cross-Validation

Here we note a few additional interesting nuances for cross-validation, for the interested reader. The main text already had a lot, and so this additional reasoning on bias and variance is relegated here.

First, you may be wondering why the bias of our estimator was not impacted by the fact that we have correlated error estimates. Usually, we take the sample average of i.i.d. samples, but here we take the sample average of correlated samples. Each $\mathsf{err}^{(j)}$ may be correlated with another $\mathsf{err}^{(i)}$ because they share data. However, this correlation does not affect the bias because $\mathbb{E}[\bar{G}] = \frac{1}{k}\sum_{j=1}^k \mathbb{E}[\mathsf{err}^{(j)}]$.

We discussed ways to reduce the variance of the CV estimator, but it is important to keep in mind that reducing the variance alone is insufficient, because the estimator is biased. For example, imagine our goal is to estimate $\mathbb{E}[\bar{G}|\mathcal{D}]$. Namely, we want to remove all stochasticity due to the resampling procedure. We can do so by significantly increasing k, where this sample average will reasonably quickly approach $\mathbb{E}[\bar{G}|\mathcal{D}]$. Even if we reduce the variance due to resampling to zero, we obtain $\mathbb{E}[\bar{G}|\mathcal{D}]$ which is not equal to $\text{GE}(f_{\mathcal{D}})$. In addition to being expensive, one recommendation is to avoid setting k too large because the confidence intervals for an interim k are more likely to include $\text{GE}(f_{\mathcal{D}})$ [28].

As one other nuanced point, notice that $\bar{G}$ can also be used as an estimator for a different quantity: the expected $\text{GE}(f_{\mathcal{D}})$ across datasets, i.e., $\mathbb{E}[(f_{\mathcal{D}}(X)-Y)^2]$ instead of $\mathbb{E}[(f_{\mathcal{D}}(X)-Y)^2|\mathcal{D}]$. We usually care about how our model, learned on our dataset, will perform in deployment, and so care about $\mathbb{E}[(f_{\mathcal{D}}(X)-Y)^2|\mathcal{D}]$. However, in other cases, we might be interested in understanding how the algorithm performs, regardless of the specific dataset, if we are interested in understanding which algorithms are more effective for the problem or related problems. There is some evidence that $\bar{G}$ is actually a better estimator

of $\mathbb{E}[(f_{\mathcal{D}}(X) - Y)^2]$, rather than of $\mathbb{E}[(f_{\mathcal{D}}(X) - Y)^2 | \mathcal{D}]$ [4]. Nonetheless, this does not mean that $\bar{G}$ is a poor estimator for $\mathrm{GE}(f_{\mathcal{D}})$, and cross validation remains a common approach for this goal.

A.3 More on GLMs

In the main text we motivated using $g = a'$ results in nice loss functions. We explain in this section. The common setting of $g = a'$ for GLMs has a connection to widely used objectives called *Bregman divergences*. These divergences are written as $D_a(\hat{y}||y)$, indicating the difference between $\hat{y}$ and y, where the divergence is parametrized by a. The minimization of this Bregman divergence corresponds to the minimization of the negative log-likelihood of the corresponding natural exponential family:

$$\operatorname*{argmin}_{\theta} D_a(x||g^{-1}(\theta)) = \operatorname*{argmin}_{\theta} -\ln p(x|\theta).$$

See [30, Section 2.2] and [1] for more details about this relationship.

Bregman divergences have nice properties, including being convex in the first argument. By selecting $g = a'$, we inherit these properties. Other choices are possible, but the resulting loss functions are likely nonconvex and will not be as well understood.

A.4 More on Constrained Optimization

A.4.1 Detailed Steps for the Proximal Update

Here we write the explicit steps we omitted in the main text. We can write the proximal update using the same expansion on c, in addition to including r

$$\begin{aligned}
\mathbf{w}_{t+1} &= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} c(\mathbf{w}_t) + \nabla c(\mathbf{w}_t)^\top(\mathbf{w} - \mathbf{w}_t) + \frac{1}{2\eta_t}\|\mathbf{w} - \mathbf{w}_t\|_2^2 + r(\mathbf{w}) \\
&= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} \nabla c(\mathbf{w}_t)^\top(\mathbf{w} - \mathbf{w}_t) + \frac{1}{2\eta_t}\|\mathbf{w} - \mathbf{w}_t\|_2^2 + r(\mathbf{w}) && \triangleright \text{ dropped constant} \\
&= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} \eta_t\nabla c(\mathbf{w}_t)^\top(\mathbf{w} - \mathbf{w}_t) + \tfrac{1}{2}\|\mathbf{w} - \mathbf{w}_t\|_2^2 + \eta_t r(\mathbf{w}) && \triangleright \text{ multiply by } \eta_t \\
&= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} \tfrac{1}{2}\|\mathbf{w} - (\mathbf{w}_t - \eta_t\nabla c(\mathbf{w}_t))\|_2^2 + \eta_t r(\mathbf{w})
\end{aligned}$$

where the last equality follows using the following facts. Let $\mathbf{a} = \mathbf{w} - \mathbf{w}_t$ and $\mathbf{b} = \eta_t\nabla c(\mathbf{w}_t)^\top$. Then we can write $\frac{1}{2}\|\mathbf{w} - (\mathbf{w}_t - \eta_t\nabla c(\mathbf{w}_t))\|_2^2 = \frac{1}{2}\|(\mathbf{w} - \mathbf{w}_t) + \eta_t\nabla c(\mathbf{w}_t))\|_2^2 = \frac{1}{2}\|\mathbf{a} + \mathbf{b}\|_2^2$. Now we can also see that $\frac{1}{2}\|\mathbf{a} + \mathbf{b}\|_2^2 = \frac{1}{2}\mathbf{a}^\top\mathbf{a} + \mathbf{a}^\top\mathbf{b} + \frac{1}{2}\mathbf{b}^\top\mathbf{b}$. Here $\mathbf{b}$ does not depend on $\mathbf{w}$, and so the minimization can drop $\frac{1}{2}\mathbf{b}^\top\mathbf{b}$.

A.4.2 Beyond Closed-form Proximal Operators

To handle one useful constraint, namely the simplex constraint, we will have to move beyond the restriction that r allows for a closed form proximal operator. The simplex constraints are those on probabilities: $\mathcal{F} = \{\mathbf{w} \in \mathbb{R}^d : \sum_{j=1}^d w_j = 1 \text{ and } w_j \geq 0\}$. To compute the second step—the projection step—we need to solve a simple optimization. To do so, we are

going to convert this constrained optimization into an unconstrained optimization on $\mathbf{w}$, by introducing additional variables. The variables are called KKT multipliers; the reason for this name will become clear after introducing the optimization.

Let $a \in \mathbb{R}$ be the KKT multiplier for the equality constraint $\sum_{j=1}^{d} w_j = 1$ and $b_j \geq 0$ the KKT multiplier for the inequality constraint $w_j \geq 0$, with vector $\mathbf{b}$ composed of b_j for $j = 1, \ldots, d$. Let $l(\mathbf{w}) \stackrel{\text{def}}{=} \frac{1}{2}\|\mathbf{w} - \mathbf{v}\|_2^2$ be the loss used in the proximal operator, where the goal is to find the closest point to a given input $\mathbf{v}$ under the constraints given by r (which here are simplex constraints). The augmented optimization problem with these additional variables is

$$\min_{\mathbf{w}\in\mathbb{R}^d} \max_{a\in\mathbb{R},\mathbf{b}\geq\mathbf{0}} L(\mathbf{w}, a, \mathbf{b}) \qquad \text{where } L(\mathbf{w}, a, \mathbf{b}) \stackrel{\text{def}}{=} l(\mathbf{w}) + a\left(\sum_{j=1}^{d} w_j - 1\right) - \sum_{j=1}^{d} b_j w_j.$$

For any $\mathbf{w}$ that satisfy the constraints, we have $l(\mathbf{w}) = \max_{a\in\mathbb{R},\mathbf{b}\geq\mathbf{0}} L(\mathbf{w}, a, \mathbf{b})$, and so optimizing for $\mathbf{w}$ as well as these additional variables results in the same optimal $\mathbf{w}$ while enforcing the constraints. To understand why, consider the possible optimal choices for a and $\mathbf{b}$. If $\mathbf{w}$ does not satisfy $\sum_{j=1}^{d} w_j - 1 = 0$, then a can be selected to make the loss arbitrarily big. For example, if $\sum_{j=1}^{d} w_j = 0.8$ and so $\sum_{j=1}^{d} w_j - 1 = -0.2$, a can be made a very large negative number, say -10^6, to add $0.2 \cdot 10^6$ to the loss. Consequently, this $\mathbf{w}$ is unlikely to result in a minimal value. Instead, for any $\mathbf{w}$ that do satisfy $\sum_{j=1}^{d} w_j - 1 = 0$, a is multiplied by zero and so cannot cause the loss to become very big; therefore, $\mathbf{w}$ will be chosen to satisfy this constraint.

This is similarly the case for $\mathbf{b}$. For any $w_j < 0$, the maximization will choose a very large b_j, producing a very high magnitude $-b_j w_j$ and resulting in the addition of a very large positive number. Again, w_j will be chosen to be ≥ 0 to avoid this situation, and the best b_j can do is to have $b_j w_k = 0$.

Now let us use this to derive an algorithm to find $\mathbf{w}$ that satisfies the constraints, for loss $l(\mathbf{w}) \stackrel{\text{def}}{=} \frac{1}{2}\|\mathbf{w} - \mathbf{v}\|_2^2$. As usual, we need to find $\mathbf{w}$ that is a stationary point

$$\begin{aligned}
\mathbf{0} = \nabla_{\mathbf{w}} L(\mathbf{w}, a, \mathbf{b}) \qquad \text{where } \nabla_{\mathbf{w}} L(\mathbf{w}, a, \mathbf{b}) &= \nabla_{\mathbf{w}} l(\mathbf{w}) + a\nabla_{\mathbf{w}}\left(\sum_{j=1}^{d} w_j - 1\right) - \nabla_{\mathbf{w}} \sum_{j=1}^{d} b_j w_j \\
&= \nabla_{\mathbf{w}} l(\mathbf{w}) + a\nabla_{\mathbf{w}}\left(\mathbf{1}^\top \mathbf{w} - 1\right) - \nabla_{\mathbf{w}} \mathbf{b}^\top \mathbf{w} \\
&= (\mathbf{w} - \mathbf{v}) + a\mathbf{1} - \mathbf{b} \\
\implies \mathbf{w} &= \mathbf{v} - a\mathbf{1} + \mathbf{b} \qquad \text{(A.2)}
\end{aligned}$$

This formula for $\mathbf{w}$ depends on the values of a and $\mathbf{b}$. We know that for a valid solution $\mathbf{w}$ in the constraint set, the best that $\mathbf{b}$ can do is to satisfy $b_j w_j = 0$ with $b_j \geq 0$. Further, we know that if $\mathbf{w}$ satisfies the constraints, a multiples by zero and cannot impact the L. For the below, we can assume a is essentially a free variable that lets us produce a valid solution for $\mathbf{w}$.

To satisfy $b_j w_j = 0$ with $b_j \geq 0$, we have to have either (a) $b_j = 0$ or (b) $w_j = 0$ with $b_j \geq 0$. We can use this to reason about the entries of the solution in Equation (A.2). For the case where $w_j \neq 0$, we must have $b_j = 0$; plugging this into Equation (A.2), we therefore have that $w_j = v_j - a$. For the second case where $w_j = 0$, b_j can be any nonnegative number. To infer what it is, we can use $w_j = v_j - a + b_j = 0$ and so $b_j = a - v_j$. For this to be

viable, this means that $a - v_j \geq 0$ and so $v_j - a < 0$. We can equivalently write that for this second case $w_j = \max(v_j - a, 0)$ because this will evaluate to zero. Similarly, for the first case, we must select an a such that $v_j - a \geq 0$. Therefore, for both cases, we have $w_j = \max(v_j - a, 0)$, assuming that we chose a appropriately.

Now we simply have to find a such that we satisfy the constraint $\mathbf{1}^\top \mathbf{w} = 1$. We actually cannot get a closed form solution for this. Instead, we have to solve an optimization to find this a, namely solve for a such that $\mathbf{1}^\top \max(\mathbf{v} - a\mathbf{1}, 0) - 1 = 0$. This is a relatively simple root finding problem, with available implementations in most packages. Note that we know there exists a feasible solution for this problem. We can start a at the maximal entry in $\mathbf{v}$ and slowly decrease it. This makes the sum smoothly increase until it equals 1.

A.5 More on Latent Factors

A.5.1 More on Sparse Coding

One strategy to obtain sparse representations is to use a sparse regularizer on the learned representation $\mathbf{h}$. This corresponds to the optimization

$$\min_{\mathbf{D}\in\mathbb{R}^{p\times d},\mathbf{H}\in\mathbb{R}^{n\times p}} \|\mathbf{X} - \mathbf{H}\mathbf{D}\|_F^2 + \lambda \sum_{i=1}^{p} \|\mathbf{H}_{:i}\|_1 + \lambda \sum_{i=1}^{p} \|\mathbf{D}_{i:}\|_2^2$$

As discussed in Section 3.2.2, the ℓ_1 regularizer promotes zeroed entries, and so prefers $\mathbf{H}$ with as many zeros as possible. A regularizer is also added to $\mathbf{D}$, to ensure that $\mathbf{D}$ does not become too large; otherwise, all the weight in $\mathbf{D}\mathbf{H}$ would be shifted to $\mathbf{D}$.

Exercise 43: Explain why all the weight in $\mathbf{D}\mathbf{H}$ would be shifted to $\mathbf{D}$, if we did not use a regularizer on $\mathbf{D}$ as well as $\mathbf{H}$. Consider what this means for identifiability, namely for uniqueness of the solution, without any regularizers. □

A.6 More on Backpropagation

First, we take the partial derivative w.r.t. the parameters $\mathbf{W}^{(1)}$.

$$\begin{aligned}
\frac{\partial c(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}^{(1)}_{jk}} &= \frac{\partial L(f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)}), \mathbf{y})}{\partial \mathbf{W}^{(1)}_{jk}} \\
&= \left(\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k}\right) \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{W}^{(1)}_{jk}} \qquad \triangleright\, \hat{\mathbf{y}}_k = f_1(\mathbf{h}\mathbf{W}^{(1)}_{:k})
\end{aligned}$$

where only $\hat{\mathbf{y}}_k$ is affected by $\mathbf{W}^{(1)}_{jk}$ in the loss, and so the gradient for the others is zero. Continuing,

$$\begin{aligned}
\frac{\partial \mathrm{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}^{(1)}_{jk}} &= \left(\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k}\right) \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \frac{\partial \boldsymbol{\theta}^{(1)}_k}{\partial \mathbf{W}^{(1)}_{jk}} \qquad \triangleright\, \boldsymbol{\theta}^{(1)}_k = \mathbf{h}\mathbf{W}^{(1)}_{:k} \\
&= \left(\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k}\right) \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \mathbf{h}_j
\end{aligned}$$

At this point these equations are abstract; but they are simple to compute for the losses and transfers we have examined. For example, for $L(\hat{\mathbf{y}}_k, \mathbf{y}_k) = \frac{1}{2}(\hat{\mathbf{y}}_k - \mathbf{y}_k)^2$, and f_1 the identity, we get

$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} = (\hat{\mathbf{y}}_k - \mathbf{y}_k) \qquad \text{and} \qquad \frac{\partial f_1(\boldsymbol{\theta}_k^{(1)})}{\partial \boldsymbol{\theta}_k^{(1)}} = 1$$

giving

$$\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}_{jk}^{(1)}} = \left(\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k}\right) \frac{\partial f_1(\boldsymbol{\theta}_k^{(1)})}{\partial \boldsymbol{\theta}_k^{(1)}} \mathbf{h}_j = (\hat{\mathbf{y}}_k - \mathbf{y}_k)\mathbf{h}_j.$$

The gradient update is as usual with $\mathbf{W}^{(1)} = \mathbf{W}^{(1)} - \alpha(\hat{\mathbf{y}} - \mathbf{y})\mathbf{h}^\top$ for some step-size α.

Next, we compute the partial gradient with respect to $\mathbf{W}^{(2)}$. Now, however, the entire output variable $\mathbf{y} \in \mathbb{R}^{1\times m}$ is affected by the choice of $\mathbf{W}_{ij}^{(2)}$ for all $i \in \{1, \ldots, p_2\}$, $j \in \{1, \ldots, p_1\}$, where for this exam. Therefore, we need to take the partial derivative w.r.t. all of $\mathbf{y}$.

$$\begin{aligned}
\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} &= \frac{\partial \sum_{k=1}^m L(f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}_{:k}^{(1)}), \mathbf{y}_k)}{\partial \mathbf{W}_{ij}^{(2)}} \\
&= \sum_{k=1}^m \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{W}_{ij}^{(2)}} && \triangleright\ \hat{\mathbf{y}}_k = f_1(\mathbf{h}\mathbf{W}_{:k}^{(1)}) = f_1(\boldsymbol{\theta}_k^{(1)}) \\
&= \sum_{k=1}^m \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}_k^{(1)})}{\partial \boldsymbol{\theta}_k^{(1)}} \frac{\partial \boldsymbol{\theta}_k^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}}.
\end{aligned}$$

Continuing,

$$\begin{aligned}
\frac{\partial \boldsymbol{\theta}_k^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} = \frac{\partial \mathbf{h}\mathbf{W}_{:k}^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} &= \frac{\partial \sum_{l=1}^p \mathbf{h}_l \mathbf{W}_{lk}^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} \\
&= \frac{\partial \sum_{l=1}^p f_2(\mathbf{x}\mathbf{W}_{:l}^{(2)})\mathbf{W}_{lk}^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} \\
&= \sum_{l=1}^p \mathbf{W}_{lk}^{(1)} \frac{\partial f_2(\mathbf{x}\mathbf{W}_{:l}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} \\
&= \mathbf{W}_{jk}^{(1)} \frac{\partial f_2(\mathbf{x}\mathbf{W}_{:j}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}}
\end{aligned}$$

because $\frac{\partial f_2(\mathbf{x}\mathbf{W}_{:l}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} = 0$ for $l \neq j$. Now continuing the chain rule

$$\begin{aligned}
\frac{\partial f_2(\mathbf{x}\mathbf{W}_{:j}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} &= \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}} \frac{\partial \boldsymbol{\theta}_j^{(2)}}{\partial \mathbf{W}_{ij}^{(2)}} && \triangleright\ \boldsymbol{\theta}_j^{(2)} = \mathbf{x}\mathbf{W}_{:j}^{(2)} \\
&= \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}} \mathbf{x}_i.
\end{aligned}$$

Putting this back together, we get

$$\begin{aligned}\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} &= \sum_{k=1}^{m} \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}_k^{(1)})}{\partial \boldsymbol{\theta}_k^{(1)}} \frac{\partial \boldsymbol{\theta}_k^{(1)}}{\partial \mathbf{W}_{ij}^{(2)}} \\ &= \sum_{k=1}^{m} \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}_k^{(1)})}{\partial \boldsymbol{\theta}_k^{(1)}} \mathbf{W}_{jk}^{(1)} \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}} \mathbf{x}_i.\end{aligned}$$

Notice that some of the gradient is the same as for $\mathbf{W}^{(1)}$, i.e.

$$\boldsymbol{\delta}_k^{(1)} = \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}_k^{(1)})}{\partial \boldsymbol{\theta}_k^{(1)}}$$

Computing these components only needs to be done once for $\mathbf{W}^{(1)}$, and this information propagated back to get the gradient for $\mathbf{W}^{(2)}$. The difference is in the gradient $\frac{\partial \boldsymbol{\theta}^{(1)}}{\partial \mathbf{W}^{(2)}}$, because $\mathbf{h}$ relies on $\mathbf{W}^{(2)}$. For $\mathbf{W}^{(1)}$, $\mathbf{h} = f_2(\mathbf{x}_i \mathbf{W}^{(2)})$ is a constant, and so does not affect the gradient for $\mathbf{W}^{(1)}$. The final gradient is

$$\begin{aligned}\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} &= \left(\sum_{k=1}^{m} \boldsymbol{\delta}_k^{(1)} \mathbf{W}_{jk}^{(1)} \right) \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}} \mathbf{x}_i \\ &= \left(\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)} \right) \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}} \mathbf{x}_i\end{aligned}$$

If another layer is added before $\mathbf{W}^{(2)}$, then the information propagated backward is

$$\boldsymbol{\delta}_j^{(2)} = \left(\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)} \right) \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}}$$

and $\mathbf{x}_i$ is replaced with $\mathbf{h}_i^{(2)}$. The gradient for $\mathbf{W}_{ij}^{(3)}$ is

$$\left(\mathbf{W}_{j:}^{(2)} \boldsymbol{\delta}^{(2)} \right) \frac{\partial f_3(\boldsymbol{\theta}_j^{(3)})}{\partial \boldsymbol{\theta}_j^{(3)}} \mathbf{x}_i$$

Example 18: Let $p(y = 1|\mathbf{x})$ be a Bernoulli distribution, with f_1 and f_2 both sigmoid functions. The loss is the cross-entropy. We can derive the two-layer update rule with these

settings, by plugging-in above.

$$
\begin{aligned}
L(\hat{y}, y) &= -y \log(\hat{y}) - (1-y)\log(1-\hat{y}) && \triangleright \text{ cross-entropy} \\
\frac{\partial L(\hat{y}, y)}{\partial \hat{y}} &= -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \\
f_2(\mathbf{x}\mathbf{W}^{(2)}_{:j}) &= \sigma(\mathbf{x}\mathbf{W}^{(2)}_{:j}) = \frac{1}{1+\exp(-\mathbf{x}\mathbf{W}^{(2)}_{:j})} \\
f_1(\mathbf{h}\mathbf{W}^{(1)}_{:k}) &= \sigma(\mathbf{h}\mathbf{W}^{(1)}_{:k}) = \frac{1}{1+\exp(-\mathbf{h}\mathbf{W}^{(1)}_{:k})} \\
\partial\sigma(\theta) &= \sigma(\theta)(1-\sigma(\theta))
\end{aligned}
$$

We can compute the backpropagation update by first propagating forward and computing

$$
\mathbf{h} = \sigma(\mathbf{x}\mathbf{W}^{(2)}) \qquad \text{and} \qquad \hat{\mathbf{y}} = \sigma(\mathbf{h}\mathbf{W}^{(1)})
$$

and then propagating the gradient back

$$
\begin{aligned}
\boldsymbol{\delta}^{(1)}_k &= \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \\
&= \left(-\frac{\mathbf{y}_k}{\hat{\mathbf{y}}_k} + \frac{1-\mathbf{y}_k}{1-\hat{\mathbf{y}}_k}\right) \hat{\mathbf{y}}_k(1-\hat{\mathbf{y}}_k) = -\mathbf{y}_k(1-\hat{\mathbf{y}}_k) + (1-\mathbf{y}_k)\hat{\mathbf{y}}_k \\
&= \hat{\mathbf{y}}_k - \mathbf{y}_k
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{W}^{(1)}_{jk}} &= \boldsymbol{\delta}^{(1)}_k \mathbf{h}_j \\
\boldsymbol{\delta}^{(2)}_j &= \left(\mathbf{W}^{(1)}_{j:}\boldsymbol{\delta}^{(1)}\right)\mathbf{h}_j(1-\mathbf{h}_j) \\
\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{W}^{(2)}_{ij}} &= \boldsymbol{\delta}^{(2)}_j \mathbf{x}_i
\end{aligned}
$$

The update simply consists of stepping in the direction of these gradients, as is usual for gradient descent. We start with some initial $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ (say filled with random values), and then apply the gradient descent rules with these gradients. □

A.7 More on Mixture Models and EM

A.7.1 Setting Up for the EM Algorithm

The key part of the EM algorithm is to explicitly reason about the latent variable z that corresponds to which mixture component generated the datapoint. To understand this, consider how data is generated from a mixture model $p(\mathbf{x}) = \sum_{k=1}^{m} w_k p(\mathbf{x}|\boldsymbol{\theta}_k)$.

1. Sample a component $z \in \{1, \ldots, m\}$ proportionally to weights $w_1, \ldots, w_m$.
2. Sample $\mathbf{x}$ from the z-th component $p(\mathbf{x}|\boldsymbol{\theta}_z)$.

In other words, we can think of w_k as $p(Z = k)$ and $p(\mathbf{x}|\boldsymbol{\theta}_k)$ as $p(\mathbf{x}|Z = k)$. When we sample $\mathbf{x}$, we can think of this as jointly sampling $(\mathbf{x}, z)$ from $p(\mathbf{x}, z) = p(\mathbf{x}|z)p(z) = p(\mathbf{x}|Z = z)p(Z = z)$, and then only observing $\mathbf{x}$. In fact, to be very explicit about this, we will now index with z instead of k and write:

$$p(\mathbf{x}) = \sum_{z=1}^{m} w_z p(\mathbf{x}|\boldsymbol{\theta}_z).$$

This is equivalent to using k, since it is just a variable name, but it forces us to think of this index as a latent variable z.

If we knew the mixture component z for $\mathbf{x}$, then estimation would be simpler. Intuitively, we can see why this is the case. We can partition the dataset into $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_m$ where $\mathcal{D}_z$ consists of the $\mathbf{x}_i$ that have label z. Then we just need to estimate the multivariate Gaussian parameters $\boldsymbol{\theta}_z = (\boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z)$ for each dataset $\mathcal{D}_z$ separately, to get the m Gaussians in the mixture. We saw how to use MLE to get these in the last section. It is a bit more work to get the coefficients w_z, especially because this is a constrained optimization, but the update ends up being simple (as we will see).

The EM algorithm is build on this insight. It iteratively predicts which component is likely for each $\mathbf{x}$ using its current parameters $\boldsymbol{\theta}^{(t)}$ (current belief about how the world works), and then uses this prediction to improve the parameter estimates $\boldsymbol{\theta}^{(t+1)} = (w_1^{(t+1)}, w_2^{(t+1)}, \ldots, w_m^{(t+1)}, \boldsymbol{\theta}_1^{(t+1)}, \ldots, \boldsymbol{\theta}_m^{(t+1)})$. The EM algorithm performs the following steps:

1. E-step: Compute $p(\mathbf{z}|\mathcal{D}, \boldsymbol{\theta}^{(t)})$ where $\mathbf{z} \stackrel{\text{def}}{=} (z_1, \ldots, z_n)$
2. M-step: Compute $\boldsymbol{\theta}^{(t+1)}$

Let us now reason about the objective we are optimizing using this procedure. We know we want to minimize the negative log-likelihood, which we can rewrite

$$\ln p(\mathcal{D}|\boldsymbol{\theta}) = \ln(p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta})/p(\mathbf{z}|\mathcal{D}, \boldsymbol{\theta})) = \ln p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}) - \ln p(\mathbf{z}|\mathcal{D}, \boldsymbol{\theta})$$

We can define

$$p_t(z_i) \stackrel{\text{def}}{=} p_t(z_i|\mathbf{x}_i, \boldsymbol{\theta}^{(t)}) \quad \text{and} \quad p_t(\mathbf{z}) \stackrel{\text{def}}{=} \prod_{i=1}^{n} p_t(z_i)$$

and take the expectation over both sides, according to the distribution p_t, giving

$$\sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathcal{D}|\boldsymbol{\theta}) = \ln p(\mathcal{D}|\boldsymbol{\theta}) \quad \text{and} \quad \sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}) - \sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathbf{z}|\mathcal{D}, \boldsymbol{\theta})$$
$$\implies \ln p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}) - \sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathbf{z}|\mathcal{D}, \boldsymbol{\theta})$$

In the maximization step of EM, we only focus on optimizing the first term, and omit $\sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathbf{z}|\mathcal{D}, \boldsymbol{\theta})$. This is better justified in Section 12.3. For now, we will move forward and derive the EM algorithm accepting that this choice still helps us minimize $-\ln p(\mathcal{D}|\boldsymbol{\theta})$.

Therefore, to get the new parameters $\boldsymbol{\theta}^{(t+1)}$, we find

$$\boldsymbol{\theta}^{(t+1)} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \; -\sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}). \tag{A.3}$$

A.7.2 The Expectation-Maximization Algorithm

Given this set-up, we can now derive the E-step and the M-step. We will do this specifically for Gaussian mixture components to start. We start with the M-step, then the E-step and finally put it all together.

M-step

Let us rewrite the objective in Equation A.3.

$$\begin{aligned}
\sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}) &= \sum_{\mathbf{z}} p_t(\mathbf{z}) \sum_{i=1}^{n} \ln p(\mathbf{x}_i, z_i|\boldsymbol{\theta}) && \triangleright\, p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}) = \Pi_{i=1}^{n} p(\mathbf{x}_i, z_i|\boldsymbol{\theta}) \\
&= \sum_{i=1}^{n} \sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathbf{x}_i, z_i|\boldsymbol{\theta}) && \triangleright\ \text{swap sums} \\
&= \sum_{i=1}^{n} \sum_{z_i=1}^{m} \sum_{\mathbf{z}_{\setminus i}} p_t(z_i) p_t(\mathbf{z}_{\setminus i}) \ln p(\mathbf{x}_i, z_i|\boldsymbol{\theta}) && \triangleright\, p_t(\mathbf{z}) = \prod_{i=1}^{n} p_t(z_i) = p_t(z_i) p_t(\mathbf{z}_{\setminus i}) \\
&= \sum_{i=1}^{n} \sum_{z_i=1}^{m} p_t(z_i) \ln p(\mathbf{x}_i, z_i|\boldsymbol{\theta}) && \triangleright \sum_{\mathbf{z}_{\setminus i}} p_t(\mathbf{z}_{\setminus i}) = 1 \text{ factors out} \\
&= \sum_{i=1}^{n} \sum_{z_i=1}^{m} p_t(z_i) \ln[w_{z_i} p(\mathbf{x}_i|\boldsymbol{\theta}_{z_i})] && \triangleright\, p(\mathbf{x}_i, z_i|\boldsymbol{\theta}) = p(\mathbf{x}_i|z_i, \boldsymbol{\theta}) p(z_i|\boldsymbol{\theta}).
\end{aligned}$$

Notice that

$$\ln[w_{z_i} p(\mathbf{x}_i|\boldsymbol{\theta}_{z_i})] = \ln w_{z_i} + \ln p(\mathbf{x}_i|\boldsymbol{\theta}_{z_i})$$

and so we can optimize for $\mathbf{w}$ and $\{\boldsymbol{\theta}_k\}_{j=1}^{m}$ separately because they do not interact. The weights $\mathbf{w}$ are in the first sum and the Gaussian parameters $\{\boldsymbol{\theta}_k\}_{j=1}^{m}$ are in the second sum.

$$\begin{aligned}
\sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}) &= \sum_{i=1}^{n} \sum_{z_i=1}^{m} p_t(z_i) \ln w_{z_i} + \sum_{i=1}^{n} \sum_{z_i=1}^{m} p_t(z_i) \ln p(\mathbf{x}_i|\boldsymbol{\theta}_{z_i}) \\
\implies \operatorname*{argmin}_{\mathbf{w} \in \mathcal{F}} - \sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}) &= \operatorname*{argmin}_{\mathbf{w} \in \mathcal{F}} - \sum_{i=1}^{n} \sum_{z_i=1}^{m} p_t(z_i) \ln w_{z_i} \\
\operatorname*{argmin}_{\boldsymbol{\theta}_k} - \sum_{\mathbf{z}} p_t(\mathbf{z}) \ln p(\mathcal{D}, \mathbf{z}|\boldsymbol{\theta}) &= \operatorname*{argmin}_{\boldsymbol{\theta}_k} - \sum_{i=1}^{n} \sum_{z_i=1}^{m} p_t(z_i) \ln p(\mathbf{x}_i|\boldsymbol{\theta}_{z_i})
\end{aligned}$$

Notice that $p_t(z_i)$ is simply stored as a multi-dimensional array $p_t[i, k]$ where at time step t we have the probabilities for z for each sample i. To derive the updates for these optimizations, we will go back to indexing by k, now that we understand the objective and that each k is a latent component.

Finding Gaussian parameters $\boldsymbol{\theta}_k$ Let us start with optimizing for the Gaussian parameters. Because

$$\sum_{i=1}^{n} \sum_{k=1}^{m} p_t[i, k] \ln p(\mathbf{x}_i|\boldsymbol{\theta}_k) = \sum_{k=1}^{m} \left(\sum_{i=1}^{n} p_t[i, k] \ln p(\mathbf{x}_i|\boldsymbol{\theta}_k) \right)$$

again we can see that each $\boldsymbol{\theta}_k$ can be optimized independently

$$\boldsymbol{\theta}_k^{(t+1)} = \underset{\boldsymbol{\theta}_k}{\operatorname{argmin}} - \sum_{i=1}^{n} p_t[i,k] \ln p(\mathbf{x}_i|\boldsymbol{\theta}_k).$$

This optimization is just a weighted log likelihood. For a Gaussian, where $\boldsymbol{\theta}_k = (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, we can show that this corresponds to finding

$$\nabla_{\boldsymbol{\mu}} - \sum_{i=1}^{n} p_t[i,k] \ln p(\mathbf{x}_i|\boldsymbol{\theta}_k) = \boldsymbol{\Sigma}_k^{-1} \sum_{i=1}^{n} p_t[i,k](\boldsymbol{\mu} - \mathbf{x}_i) = \mathbf{0} \implies \sum_{i=1}^{n} p_t[i,k](\boldsymbol{\mu} - \mathbf{x}_i) = \mathbf{0}$$
$$\implies \boldsymbol{\mu} = \frac{1}{\sum_{i=1}^{n} p_t[i,k]} \sum_{i=1}^{n} p_t[i,k]\mathbf{x}_i$$

The normalization takes into account how much probability is associated with component k across the samples. We can further simplify this by using a normalized weighting

$$\tilde{p}_t[i,k] = \frac{p_t[i,k]}{\sum_{j=1}^{n} p_t[j,k]}$$

to get $\boldsymbol{\mu}_k = \sum_{i=1}^{n} \tilde{p}_t[i,k]\mathbf{x}_i$. Similarly, for the variance, we find that

$$\boldsymbol{\Sigma}_k = \sum_{i=1}^{n} \tilde{p}_t[i,k](\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top.$$

Finding component probabilities w Now let us turn to finding $\mathbf{w}$. (Note: we already solved this in Section 6.3, but repeat here to match the above terminology). This optimization is a bit more complex, because we have to ensure that we satisfy the simplex constraints. In other words, we need to find

$$\mathbf{w}^{(t+1)} = \underset{\mathbf{w} \in [0,1]^m, \sum_{k=1}^{m} w_k = 1}{\operatorname{argmin}} - \sum_{i=1}^{n} \sum_{k=1}^{m} p_t[i,k] \ln w_k$$

Overloading terminology, if we let $p_t[k] = \sum_{i=1}^{n} p_t[i,k]$, and notice $\sum_{i=1}^{n} \sum_{k=1}^{m} p_t[i,k] \ln w_k = \sum_{k=1}^{m} \ln w_k \sum_{i=1}^{n} p_t[i,k] = \sum_{k=1}^{m} \ln w_k p_t[k]$, then we can equivalently write

$$\mathbf{w}^{(t+1)} = \underset{\mathbf{w} \in [0,1]^m, \sum_{k=1}^{m} w_k = 1}{\operatorname{argmin}} - \sum_{k=1}^{m} p_t[k] \ln w_k$$

We can use our new-found knowledge about constrained optimization and KKT multipliers, for this problem. To start we shall first form the Lagrangian function as

$$L(\mathbf{w}, a, \mathbf{b}) = - \sum_{k=1}^{m} p_t[k] \ln w_k + a \left(\sum_{k=1}^{m} w_k - 1 \right) - \sum_{k=1}^{m} b_k w_k$$

where $a \in \mathbb{R}$ and $\mathbf{b} \geq \mathbf{0}$ are KKT multipliers. Our goal now is to solve the new objective that no longer has constraints on $\mathbf{w}$:

$$\min_{\mathbf{w} \in \mathbb{R}^m} \max_{a \in \mathbb{R}, \mathbf{b} \geq 0} L(\mathbf{w}, a, \mathbf{b})$$

We can find a closed form solution, by reasoning about the feasible solutions. We know that any optimal solution $\mathbf{w}$ must satisfy $\mathbf{w} \in \mathcal{F}$ (the simplex); otherwise, the loss $L(\mathbf{w}, a, \mathbf{b})$ can be made arbitrarily big by (the adversaries) a and $\mathbf{b}$. So, any optimal solution will have $\mathbf{w} \in \mathcal{F}$. Moreover, for such $\mathbf{w}$, we know that $a \in \mathbb{R}$ has no impact on the loss, since it multiples zero; therefore, it is a free variable and can be anything and still result in an optimal solution—namely result in the same value for $L(\mathbf{w}, a, \mathbf{b})$. Finally, we know that $\mathbf{b} \geq 0$ will be chosen such that $b_k w_k = 0$ for all k, since that is the choice that makes the sum involving $\mathbf{b}$ maximal. And, of course, we need to be at a stationary point for $\mathbf{w}$. In other words, to satisfy the KKT conditions and know we have an optimal solution, we need

$$\begin{aligned} 0 &= \frac{\partial}{\partial w_k} L(\mathbf{w}, a, \mathbf{b}) = -\frac{p_t[k]}{w_k} + a - b_k \qquad \forall k \in \mathcal{Y} \\ w_k b_k &= 0 \end{aligned}$$

with $\mathbf{b} \geq 0$ and $\mathbf{w} \in \mathcal{F}$. The stationarity conditions gives us $w_k = \frac{p_t[k]}{a - b_k}$. As in Section **??**, we know that $b_k = 0$ unless $w_k = 0$. If $p_t[k] > 0$, we know $w_k > 0$, giving $w_k = \frac{p_t[k]}{a}$. If $p_t[k] = 0$, then $w_k = 0$ and similarly we can write $w_k = \frac{p_t[k]}{a}$. Therefore, to ensure we are at an optimal solution—and satisfy the KKT conditions—we know we need to set a such that $\sum_{k=1}^m w_k = 1$. We can do so by setting $a = n$, giving

$$\sum_{k=1}^m w_k = \sum_{k=1}^m \frac{p_t[k]}{n} = \frac{1}{n} \sum_{k=1}^m p_t[k] = \frac{1}{n} n = 1$$

because by definition $\sum_{k=1}^m p_t[k] = n$. Therefore,

$$w_k^{(t+1)} = \frac{1}{n} p_t[k] \quad \text{where} \quad p_t[k] \stackrel{\text{def}}{=} \sum_{i=1}^n p_t[i, k] \tag{A.4}$$

E-step

Once we have the new parameters—once we have done the maximization step—we need to update the component probabilities—namely do the expectation step. This involves getting

$$p_{t+1}(Z_i = k) \stackrel{\text{def}}{=} p(Z_i = k | \mathbf{x}_i, \boldsymbol{\theta}^{(t+1)}) = \frac{w_k^{(t+1)} p(\mathbf{x}_i | \boldsymbol{\theta}_k^{(t+1)})}{\sum_{j=1}^m w_j^{(t+1)} p(\mathbf{x}_i | \boldsymbol{\theta}_j^{(t+1)})}. \tag{A.5}$$

These again can be stored as an $n \times m$ matrix $p_{t+1}[i, k]$ and the normalized $\tilde{p}_{t+1}[i, k]$ computed from those. Note that in practice, we do not create new variables p_{t+1}, but rather overwrite the existing $n \times m$ matrix; we simply index by time for clarity here.

Summarizing the EM Algorithm

In general, for each step t, the EM algorithm performs the following steps:

1. E-step: Compute $p(\mathbf{z} | \mathcal{D}, \boldsymbol{\theta}^{(t)})$
2. M-step: Compute $\boldsymbol{\theta}^{(t+1)}$

Exercise 44: Derive the updates again now assuming the mixture components are exponentials and $d = 1$. □

A.7.3 Identifiability

When estimating the parameters of a mixture, it is possible that for some parametric families one obtains multiple solutions. In other words, for all $\mathbf{x} \in \mathcal{X}$,

$$p(\mathbf{x}|\boldsymbol{\theta}) = \sum_{k=1}^{m} w_k p(\mathbf{x}|\boldsymbol{\theta}_k) = \sum_{j=1}^{m'} w'_j p(\mathbf{x}|\boldsymbol{\theta}'_j) = p(\mathbf{x}|\boldsymbol{\theta}')$$

even though $\boldsymbol{\theta} \neq \boldsymbol{\theta}'$. The parameters are *identifiable* if

$$\sum_{k=1}^{m} w_k p(\mathbf{x}|\boldsymbol{\theta}_k) = \sum_{k=1}^{m'} w'_j p(\mathbf{x}|\boldsymbol{\theta}'_k),$$

implies that $m = m'$ and that there exists a re-ordering of the parameters such that $w_k = w'_k$ and $\boldsymbol{\theta}_k = \boldsymbol{\theta}'_k$. We explicitly state that there is a re-ordering because the order of the mixing components is irrelevant, so they may need to be re-ordered to finding the matching pairs.

It is well-known that a mixture of Gaussian distributions is identifiable. In fact, more generally, mixtures of exponential family distributions—which we discuss in the next Chapter—are identifiable. Some distributions, however, may not satisfy this property.

A.7.4 Connection to Mirror Descent

The EM algorithm actually has an interesting connection to gradient descent [15]. More specifically, it has a connection to an algorithm called *mirror descent*. The idea behind mirror descent is simple: when deriving the gradient descent update, we use a different distance d to the previous parameters:

$$\mathbf{w}_{t+1} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} c(\mathbf{w}_t) + \nabla c(\mathbf{w}_t)^\top (\mathbf{w} - \mathbf{w}_t) + \frac{1}{2\eta_t} d(\mathbf{w}, \mathbf{w}_t)$$

where for gradient descent we used $d(\mathbf{w}, \mathbf{w}_t) = \|\mathbf{w} - \mathbf{w}_t\|_2^2$.

The choice of squared distance is reasonable in many cases, but not that reasonable when the parameters are those for a probability distribution, as they are in mixture models. We do not in fact care too much if the parameters are close in Euclidean space—namely according to squared error—but rather care more if their mixture distributions are similar. We have seen one useful way to measure differences between distributions, in Section 2.5: KL-divergences! We can use

$$d(\mathbf{w}, \mathbf{w}_t) = \mathrm{KL}(p(\mathbf{x}; \mathbf{w}_t) || p(\mathbf{x}; \mathbf{w}))$$

It has been shown that each iteration of EM actually corresponds to computing $\mathbf{w}_{t+1}$ with this d, with a stepsize of 1, for a reasonably large set of mixture distributions [15]. This connection is useful because it motivates the EM algorithm further, as well as suggests ways to extend the algorithm using algorithmic choices in mirror descent.

A.8 More on Generative Models

A.8.1 Contrasting with Generative Classifiers

The term generative models has also been used to describe an alternative approach to classification. The standard approach we have considered is to learn a *discriminative classifier*:

one where we learn $p(y|\mathbf{x})$. This model lets us discriminate between possible targets for a given input $\mathbf{x}$. A *generative classifier* is still trying to obtain $\text{argmax}_{y \in \mathcal{Y}} p(y|\mathbf{x})$, but instead estimates $p(\mathbf{x}|y)$ and $p(y)$. It uses the fact that

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \implies \underset{y \in \mathcal{Y}}{\text{argmax}}\, p(y|\mathbf{x}) = \underset{y \in \mathcal{Y}}{\text{argmax}}\, p(\mathbf{x}|y)p(y)$$

You can see why this is called a generative classifier, since we learn the joint distribution $p(\mathbf{x}|y)p(y) = p(\mathbf{x}, y)$ that enables us to generate (sample) pairs $(\mathbf{x}, y)$. Further, because $\mathbf{x}$ is often more complex—such as an image—we use approaches that we use for generative models, such as mixture models or variational autoencoders.

However, this approach seems to violate the principle of simplicity. It is typically easier to learn $p(y|\mathbf{x})$. Why learn a full generative model on $\mathbf{x}$ when all you really need is $p(y|\mathbf{x})$? Primarily the answers are that (a) we can encode different inductive biases (priors) into our generative models and (b) we can learn relatively simple generative models and still obtain reasonable classification accuracy. For example, if you are trying to classify faces into *narrow* or *wide*, then as an expert you might be able to encode prior knowledge into $p(\mathbf{x}|y = \text{narrow})$ and $p(\mathbf{x}|y = \text{wide})$. It can actually be harder to encode prior knowledge into $p(y|\mathbf{x})$.

For the second point, a canonical example of a generative classifier is *naive Bayes*. This algorithm is called naive because it imposes a simplistic (and likely untrue) assumption: that all the features are independent given the class information. Mathematically,

$$p(\mathbf{x}|y) = p(x_1, x_2, \ldots, x_d|y) = \prod_{j=1}^{d} p(x_j|y)$$

The utility of this (strong) assumption is that it is much simpler to learn these univariate distributions over each features, than it is to learn one larger joint distribution. If we have m classes, then we simply learn m univariate distributions over each x_j, for a total of md univariate distributions. For example, we can learn Gaussians $p(x_j|y = k) = \mathcal{N}(\mu_{jk}, \sigma^2_{jk})$ or we can even learn mixture models $p(x_j|y = k)$ for each (j, k). In either case, we have relatively simple algorithms to do so. For Gaussians, we have a closed form solution: the maximum likelihood estimates μ_{jk} and σ^2_{jk} are simply the mean and variance for feature j for each data point labeled class k. For mixture models, we can similarly create individual datasets and use our EM algorithm. We take all the points labeled as class k and include only feature j: $\mathcal{D}_{jk} = \{x_{ij} : y_i = k \text{ for } (x_i, y_i) \in \mathcal{D}\}$.

A.9 More on Generalization Theory

In this section, we talk about the basic ideas behind generalization bounds. This is a brief introduction, that is primarily to pique your interest and only scratches the surface. *Statistical learning theory* is a constantly evolving field, with many new discoveries. Many advances are also focused on more information theoretical results, under more specialized cases or assuming properties of the data. Here, we focus on the simplest case, where we assume little about the structure of the data, and consider primarily the role of the complexity of the function class.

We begin with some basic finite-sample results, that relate the complexity of the model class to the number of samples required to obtain a reasonable estimate of expected error (generalization error). We will discuss one result using *concentration inequalities* and *Rademacher complexity* to characterize model-class complexity; for further information, you could consider this tutorial on the topic [8].

A.9.1 A Shorter Overview

Most generalization bounds are focused on obtaining high-probability guarantees. The general structure is rather simple. In this section we outline this general structure, and provide more specific details for your interest in the following of sections.

Our goal is still to select a function from $\mathcal{F}$ to minimize generalization error,

$$\min_{f\in\mathcal{F}} \mathrm{GE}(f) \qquad \text{where} \qquad \mathrm{GE}(f) = \mathbb{E}[\mathrm{cost}(f(\boldsymbol{X}), Y)] = \int_{\mathcal{X}\times\mathcal{Y}} p(\mathbf{x}, y)\mathrm{cost}(f(\mathbf{x}), y)d\mathbf{x}dy.$$

We minimize a sample error as a proxy to the true expected error,

$$\widehat{\mathrm{GE}}(f) = \frac{1}{n}\sum_{i=1}^{n} \mathrm{cost}(f(\mathbf{x}_i), y_i).$$

We can decompose the true error using

$$\begin{aligned}\mathrm{GE}(f) &= \widehat{\mathrm{GE}}(f) + \mathrm{GE}(f) - \widehat{\mathrm{GE}}(f) \\ &\leq \widehat{\mathrm{GE}}(f) + \max_{h\in\mathcal{F}} \left(\mathrm{GE}(h) - \widehat{\mathrm{GE}}(h)\right).\end{aligned}$$

The term $\widehat{\mathrm{GE}}(h)$ is dependent on the data, and so is a random variable. It is hard to reason about the worst-case difference for a specific dataset. Instead, we use concentration inequalities to get an upper bound with high probability. Under certain assumptions—needed to be able to use the desired concentration inequality—we get a bound of the following form: for some constant c, with probability $1-\delta$,

$$\mathrm{GE}(f) \leq \widehat{\mathrm{GE}}(f) + \mathbb{E}_{\mathcal{D}}\left[\max_{h\in\mathcal{F}} \left(\mathrm{GE}(h) - \widehat{\mathrm{GE}}(h)\right)\right] + c\sqrt{\frac{-\ln\delta}{n}}.$$

The second term $R = \max_{h\in\mathcal{F}}(\mathrm{GE}(h) - \widehat{\mathrm{GE}}(h))$ is now upper-bounded by its expectation plus an interval to account for how far the sample R deviates from its expectation $\mathbb{E}[R]$.

This R reflects the complexity of the function class. The R is bigger if the function class is more complex, because there is some h that can overfit the dataset and so make $\widehat{\mathrm{GE}}(h)$ very small but have high true error $\mathrm{GE}(h)$. The expectation reflects this complexity across training datasets, and so can be thought of as the expected complexity.

Different bounds arise depending on the assumptions. These types of bounds are distribution-dependent, in that these assumptions typically restrict the types of data distributions that we have to make it appropriate to use the chosen concentration inequality. It is not important here to know how we measure complexity, nor specific details about these assumptions. Rather, the purpose of the above was to get an idea of how we reason about high probability generalization bounds. For specific examples of complexity measures and assumptions, see the next few sections.

A.9.2 A Generalization Bound for Linear Regression

Our goal throughout this book has been to obtain a function, based on a set of examples, that predicts accurately: produces low expected error across the space of possible examples. We cannot, however, measure the expected error. Statistically, we know that with a sufficient sample, we can approximate an expectation. Here, we quantify this more carefully for learned functions.

Our goal more precisely is to select a function from a function class $\mathcal{F}$ to minimize a loss function $\ell : \mathbb{R} \times \mathbb{R} \to [0, \infty)$ in expectation over all pairs $(\mathbf{x}, y)$

$$\min_{f \in \mathcal{F}} \mathbb{E}[\ell(f(\boldsymbol{X}), Y)].$$

For example, in linear regression, $\mathcal{F} = \{f : \mathbb{R}^d \to \mathbb{R} \mid f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}, \text{ for any } \mathbf{w} \in \mathbb{R}^d\}$. This space of functions $\mathcal{F}$ represents all possible linear functions of inputs $\mathbf{x} \in \mathbb{R}^d$, to produce a scalar output. Our goal in linear regression was to minimize a proxy to the true expected error, i.e., the sample error: $\frac{1}{n}\sum_{i=1}^n \ell(f(\mathbf{x}_i), y_i)$. Now a natural question to ask is: does this sample error provide an accurate estimate of the true expected error? And what does it tell us about the true generalization performance, i.e., true expected error?

Let's start with a simple example, using linear regression. Assume a bounded function class $\mathcal{F}$, where $\mathcal{F} = \{f : \mathbb{R}^d \to \mathbb{R} \mid f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}, \text{ for any } \mathbf{w} \in \mathbb{R}^d \text{ such that } \|\mathbf{w}\|_2 \leq B_w\}$ for some finite scalar $B_w > 0$. Assume the input features come from a bounded space, such that for all $\mathbf{x}$, $\|\mathbf{x}\|_2 \leq B_x$ for some finite scalar $B_x > 0$, and further that the outputs $y \in [-B_y, B_y]$ for some $B_y > 0$. Assume we use loss $\ell(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$. This loss is Lipschitz continuous for our bounded region, which means that the function does not change too quickly. Namely, for $\hat{y}_1, \hat{y}_2 \in [-B_y, B_y]$, and any $y \in$, there is a constant K such that

$$|\ell(\hat{y}_1, y) - \ell(\hat{y}_2, y)| \leq K|\hat{y}_1 - \hat{y}_2|.$$

This constant K reflects how fast the function can change, since it is the ratio between the change in the function for two points to the distance between those two points (rise over run). The squared error is not Lipschitz for all of $\mathbb{R}$, since it grows quickly once $\hat{y}$ gets bigger. But, for our bounded region it is Lipschitz, with Lipschitz constant $K = B_y + B_x B_w$. We can compute K because it is the maximum magnitude of the gradient of the function, wrt to its inputs $\hat{y}$. Using the fact that $|\hat{y}| \leq B_x B_w$, we have

$$\left|\frac{d\ell(\hat{y}, y)}{d\hat{y}}\right| = |\hat{y} - y| \leq |\hat{y}| + |y| \leq B_y + B_x B_w.$$

Further, because $y \in [-B_y, B_y]$, we know the loss is bounded as

$$\ell(\hat{y}, y) = \tfrac{1}{2}(\hat{y} - y)^2 \leq \tfrac{1}{2}(B_y^2 + B_x^2 B_w^2).$$

For approximate error

$$\widehat{\mathrm{Err}}(f) = \frac{1}{n}\sum_{i=1}^n \ell(f(\mathbf{x}_i), y_i)$$

and true error

$$\text{Err}(f) = \mathbb{E}[\ell(f(\boldsymbol{X}), Y)] = \int_{\mathcal{X}\times\mathcal{Y}} p(\mathbf{x}, y)\ell(f(\mathbf{x}), y)d\mathbf{x}dy$$

using Equation A.7 below, we get that with probability $1-\delta$, for $\delta \in (0, 1]$,

$$\text{Err}(f) \leq \widehat{\text{Err}}(f) + \frac{2KB_xB_w}{\sqrt{n}} + \tfrac{1}{2}(B_y^2 + B_x^2B_w^2)\sqrt{\frac{\ln(1/\delta)}{2n}}. \tag{A.6}$$

With increasing samples n, the second two terms disappear and the sample error approaches the true expected error. This bound show the rate at which this discrepancy disappears. For a higher confidence—small δ making $\ln(1/\delta)$ larger—more samples are need for the third term to be small. This third term is obtained using *concentration inequalities*, which enable us to state the rate at which a sample mean gets close to its expected value. For possibly large values of features or learned weights, the second term can be big and can again require more samples. The second term reflects the properties of our function class: a simpler class, with small bounded weights, can have a more accurate estimate of the loss on a smaller number of samples. More generally, this complexity measure is called the *Rademacher complexity*.[2] For the linear functions above, with bounded ℓ_2 norms for $\mathbf{x}$, $\mathbf{w}$, the Rademacher complexity is bounded as $R_n(\mathcal{F}) \leq B_xB_y/\sqrt{n}$ (see [13, Equation 3]).

In the next few sections, we provide a generalization result for more general functions, as well as required background to determine that result.

A.9.3 Complexity of a function class

Rademacher complexity of a function class characterizes the overfitting ability of functions, on a particular sample. Function classes that are more complex have functions that are more likely to be able to fit random noise, and so have higher Rademacher complexity. The empirical Rademacher complexity, for a sample $\{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ —where typically we consider $\mathbf{z}_i = (\mathbf{x}_i, y_i)$ — is defined as[3]

$$\hat{R}_n(\mathcal{F}) = \mathbb{E}\left[\max_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n} \sigma_i f(\mathbf{x}_i)\right]$$

where the expectation is over i.i.d.random variables $\sigma_1, \ldots, \sigma_n$ chosen uniformly from $\{-1, 1\}$. This choice reflects how well the function class can correlate with this random noise. Consider for example if $f(\mathbf{x})$ predicts 1 or -1, as in binary classification. If there exists a function in the class of functions that can perfectly match the sign of the randomly sampled σ_i, then that function produces the highest value $\sum_{i=1}^{n} \sigma_i f(\mathbf{x}_i)$. The empirical Rademacher complexity for a function class is high, if for any randomly sampled σ_i, there exists such a function within the function class (can be a different function for each $\sigma_1, \ldots, \sigma_n$). The Rademacher

[2] If you have heard of VC dimension, we will discuss the connection between Rademacher and VC dimension below. They both play a role in identifying the complexity of a function class.

[3] Here we are being a bit loose and using maximum instead of supremum, to avoid burdening the reader with new terminology. We usually deal with function classes $\mathcal{F}$ where using the supremum is equivalent to using the maximum. The supremum is used when a set does not contain a maximal point (e.g., $[0, 1)$), where the supremum provides the closest upper bound (e.g., 1 for $[0, 1)$).

complexity is the expected empirical Rademacher complexity, over all possible samples of n instances.

For function classes with high Rademacher complexity, error on the training set is unlikely to be reflective of the generalization error, until there is a sufficient number of samples. This is reflected in the generalization bound in Section A.9.4.

Connection to VC dimension: The complexity of a function class can also be characterized by the VC dimension. The idea of VC dimension is to characterize the number of points that can be separated (or shattered) by a function class. Simple functions have low VC dimension, because they are not complex enough to separate many points. More complex functions, that enable complex boundaries, have higher VC dimension. For example, for functions of the form $f((x_1, x_2)) = \text{sign}(x_1 w_1 + x_2 w_2 + w_0)$, the VC dimension is 3; more generally, for $\mathbf{x} \in \mathbb{R}^d$, the VC dimension is $d + 1$. VC dimension is a similar idea to Rademacher complexity, but it is restricted to binary classifiers. For this reason, we directly discuss the Rademacher complexity, which for binary classifiers can be bounded in terms of the VC dimension. By Sauer's Lemma, we can typically bound the Rademacher complexity of a hypothesis class by $\sqrt{\frac{2\text{VC-dimension} \ln n}{n}}$.

A.9.4 A Generalization Bound for General Function Classes

The generalization bound for a class of models can be obtained by combining the concentration inequalities to bound deviation from the mean for fewer samples, and using the Rademacher complexity to bound the difference between the sample error and true expected error across all functions in the function class. We additionally need to restrict the set of losses. We assume that the losses are Lipschitz with constant K, meaning that they do not change too quickly in a region, with c indicating the rate of change. Further, we also assume that the loss is bounded by b, i.e., attains values in $[-b, b]$. As above, if $\{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ is i.i.d., then with probability $1 - \delta$, for every $f \in \mathcal{F}$,

$$\mathbb{E}[\ell(f(\boldsymbol{X}), Y)] \leq \frac{1}{n} \sum_{i=1}^{n} \ell(f(\mathbf{x}_i), y_i) + 2K R_n(\mathcal{F}) + b\sqrt{\frac{\ln(1/\delta)}{2n}} \tag{A.7}$$

For a more precise theorem statement and a proof, see [3, Theorem 7] and [13, Theorem 1].

A.10 More on Missing Data

A.10.1 Multiple Imputation and the MAR Assumption

Consider an idealized scenario. If we could, we would have the distribution for each subset of available variables, to then allow us to compute $p(y|\mathbf{x})$ for only the available terms in $\mathbf{x}$. In other words, what we would like to do is find $p(y|\mathbf{x}_\mathcal{A})$ where $\mathbf{x}_\mathcal{A}$ are the available components of $\mathbf{x}$, and $\mathbf{x}_\mathcal{M}$ are the remaining missing ones. We can write this probability, in terms of the conditional probabilities $p(y|\mathbf{x})$, where the (unknown) complete vector $\mathbf{x}$ is

composed of $\mathbf{x}_\mathcal{A}$ and $\mathbf{x}_\mathcal{M}$.

$$\begin{aligned} p(y|\mathbf{x}_\mathcal{A}) &= \int p(y, \mathbf{x}_\mathcal{M}|\mathbf{x}_\mathcal{A}) d\mathbf{x}_\mathcal{M} && \triangleright \text{ marginalization} \\ &= \int p(y|\mathbf{x}_\mathcal{M}, \mathbf{x}_\mathcal{A}) p(\mathbf{x}_\mathcal{M}|\mathbf{x}_\mathcal{A}) d\mathbf{x}_\mathcal{M} && \triangleright \text{ chain rule} \end{aligned}$$

We could approximate this integral, which is an expected value over $\mathbf{x}_\mathcal{M}$, using a sample average. If we could sample multiple $\mathbf{x}_{\mathcal{M},1}, \ldots, \mathbf{x}_{\mathcal{M},b}$ from $p(\mathbf{x}_\mathcal{M}|\mathbf{x}_\mathcal{A})$, then we could approximate $p(y|\mathbf{x}_\mathcal{A}) \approx \frac{1}{b}\sum_{i=1}^{b} p(y|\mathbf{x}_\mathcal{A}, \mathbf{x}_{\mathcal{M},i})$. Or, if we are doing regression rather than classification, we could use $\mathbb{E}[Y|\mathbf{x}_\mathcal{A}] \approx \frac{1}{b}\sum_{i=1}^{b} \mathbb{E}[Y|\mathbf{x}_\mathcal{A}, \mathbf{x}_{\mathcal{M},i}]$ where each term $\mathbb{E}[Y|\mathbf{x}_\mathcal{A}, \mathbf{x}_{\mathcal{M},i}]$ is an output from the regression model on inputs $\mathbf{x}_\mathcal{A}, \mathbf{x}_{\mathcal{M},i}$. This approach also gives us a range of possible predictions, based on the variance across these b samples, providing some insight into how different the prediction could have been if we had seen different values for $\mathbf{x}_\mathcal{M}$. This approach is called *multiple imputation*.

We can do multiple imputation by learning a model that lets us sample $\mathbf{x}_\mathcal{M}$ for given $\mathbf{x}_\mathcal{A}$. One setting where this is straightforward to do is probabilistic PCA. In PPCA, we have Gaussian distribution over $\mathbf{x}$, $p(\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu} = \mathbf{0}, \mathbf{D}\mathbf{D}^\top + \sigma^2\mathbf{I})$, where we learned $\mathbf{D}$ and σ. For such a Gaussian, we can compute the conditional distribution $p(\mathbf{x}_\mathcal{M}|\mathbf{x}_\mathcal{A})$, which also remains Gaussian. In other words, for

$$\begin{bmatrix} \mathbf{x}_\mathcal{A} \\ \mathbf{x}_\mathcal{M} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_\mathcal{A} \\ \boldsymbol{\mu}_\mathcal{M} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{\mathcal{A}\mathcal{A}} & \boldsymbol{\Sigma}_{\mathcal{A}\mathcal{M}} \\ \boldsymbol{\Sigma}_{\mathcal{M}\mathcal{A}} & \boldsymbol{\Sigma}_{\mathcal{M}\mathcal{M}} \end{bmatrix} \right)$$

we know that

$$p(\mathbf{x}_\mathcal{M}|\mathbf{x}_\mathcal{A}) \quad \text{is} \quad \mathcal{N}\left(\boldsymbol{\mu}_\mathcal{M} + \boldsymbol{\Sigma}_{\mathcal{M}\mathcal{A}}\boldsymbol{\Sigma}_{\mathcal{A}\mathcal{A}}^{-1}(\mathbf{x}_\mathcal{A} - \boldsymbol{\mu}_\mathcal{A}),\ \boldsymbol{\Sigma}_{\mathcal{M}\mathcal{M}} - \boldsymbol{\Sigma}_{\mathcal{M}\mathcal{A}}\boldsymbol{\Sigma}_{\mathcal{A}\mathcal{A}}^{-1}\boldsymbol{\Sigma}_{\mathcal{A}\mathcal{M}}\right).$$

This formula is not simple or that easy to interpret, but it is easy to compute and sample from. We simply compute the above mean and covariance, and sample from a Gaussian to get a sample of $\mathbf{x}_\mathcal{M}$. We can do this repeatedly, to get a multiple imputation estimate for our prediction.

We can do this for other generative models too, but sampling becomes a bit more complicated and expensive. There are many algorithms out there for it, though, so it is a feasible route! The sampling literature is vast and we will not cover those approaches here.

An important part of this approach is the assumption that the data is **Missing At Random (MAR)**. This assumption states that conditioned on the observable information—other features that are not missing—the distribution over the missing feature is not skewed. In the terms above, this means we are assuming that $p(\mathbf{x}_\mathcal{M}|\mathbf{x}_\mathcal{A}) = p(\mathbf{x}_\mathcal{M}|\mathbf{x}_\mathcal{A}, \mathcal{M} \text{ is missing})$. Essentially, we are saying that knowing certain features are missing is not relevant, and this conditional distribution remains the same regardless whether those features are missing or not.

Example 19: Assume there is a true joint distribution $p(\mathbf{x})$, where $\mathbf{x}$ is information about a patient (disease severity, gender, age, etc.). For $x_2 = \texttt{male}, x_3 = \texttt{30}$, we have a distribution over $x_1 = \texttt{disease severity} \in [0, 1]$: $p(x_1|x_2 = \texttt{male}, x_3 = \texttt{30}$. This is the true distribution in the population; let's say it is centered at a severity of 0.1 for 30 year old males.

But now let's imagine that men are less likely to report any information about the disease, so this feature x_1 is more often missing for these patients. This means in the dataset, whether or not x_1 is missing will be correlated with seeing $x_2 =$ male. But, we may still be able to say it is MAR, conditioned on the fact that the patient is listed as a male: once we know that the patient is `male`, the fact that the feature is missing does not tell us anything the severity of the disease. If we assume x_1 is missing, and $\mathbf{x}_{\setminus 1}$ are the remaining features that are observed, then MAR means

$$p(x_1|\mathbf{x}_{\setminus 1}, x_1 \text{ is missing}) = p(x_1|\mathbf{x}_{\setminus 1})$$

□

This MAR assumption is likely to be violated, at least somewhat. In this example, it is not easy to know if someone with low or high severity is more or less likely to report disease information. Someone with low severity may be lazy and not bother. Someone with high severity may be unhappy about the situation and choose to not report. If there is balance across severities in terms of willingness to report—and the primary differences in whether the info is reported are due to the observed factors like gender and age—then MAR is a reasonable assumption, even if not perfectly true. In other cases, it is clear that MAR is not reasonable; and so we say that the data is Missing Not At Random (MNAR).

A.10.2 Naive Bayes and Missing Data

A benefit of generative classifiers like naive Bayes is that they allow us to deal with missing data using marginalization. Recall that in naive Bayes we classify using $p(\mathbf{x}|y)p(y)$ instead of $p(y|\mathbf{x})$, because

$$\underset{y\in\mathcal{Y}}{\operatorname{argmax}}\, p(y|\mathbf{x}) = \underset{y\in\mathcal{Y}}{\operatorname{argmax}}\, p(\mathbf{x}|y)p(y)$$

Now assume we have a point $\mathbf{x}_i$, and assume attribute 2 is missing. But, $p(\mathbf{x}|y)$ requires all of $\mathbf{x}$ to be specified. Instead, we would like to use only our available information. If we could, then we would have the learned model $p(x_{i1}, x_{i3}, x_{i4}, \ldots, x_{id}|y)$. Then to classify the input, we would instead use $\operatorname{argmax}_{y\in\mathcal{Y}} p(x_{i1}, x_{i3}, x_{i4}, \ldots, x_{id}|y)p(y)$.

Implicitly, we have learned all of these marginals, by learning the full joint distribution. We can always extract out the distribution over a subset of variables, by marginalizing over the other (missing) variables. For this example, we have that

$$p(x_1, x_3, x_4, \ldots, x_d|y) = \int_{\mathcal{X}_2} p(x_1, x_2, x_3, x_4, \ldots, x_d|y)p(y)dx_2 \int_{\mathcal{X}_2} p(\mathbf{x}|y)p(y)dx_2$$

For naive Bayes, this marginal is easy to obtain, because each feature is independent given the class. In fact, we get

$$p(x_1, x_3, x_4, \ldots, x_d|y) = p(x_1|y)p(x_3|y)\ldots p(x_d|y)$$

which is the same as

$$\begin{aligned}\int_{\mathcal{X}_2} p(x_1, x_2, x_3, x_4, \ldots, x_d|y)dx_2 &= \int_{\mathcal{X}_2} p(x_1|y)p(x_2|y)p(x_3|y)\ldots p(x_d|y)dx_2 \\ &= p(x_1|y)p(x_2|y)p(x_3|y)\ldots p(x_d|y)\int_{\mathcal{X}_2} p(x_2|y)dx_2 \\ &= p(x_1|y)p(x_2|y)p(x_3|y)\ldots p(x_d|y) \qquad \triangleright \text{ where} \int_{\mathcal{X}_2} p(x_2|y)dx_2 = 1.\end{aligned}$$

More generally, for any input $\mathbf{x}_i$ with available features $\mathcal{A}_i$, we get that

$$\underset{y\in\mathcal{Y}}{\operatorname{argmax}}\, p(\mathbf{x}_i|y)p(y) = \underset{y\in\mathcal{Y}}{\operatorname{argmax}}\, p(y)\prod_{j\in\mathcal{A}_i} p(x_{ij}|y)$$

Appendix B

Convergence Rates for Gradient Descent

We know the optimization procedure is important, but have not yet theoretically analyzed these optimization choices. Fortunately for us, some of the optimization theory in machine learning is quite simple. In this chapter, we will go through the convergence proof for gradient descent. Then we will contrast the convergence rates for our different algorithms, and particularly get a more concrete understanding of how the mini-batch size might impact convergence rates. We conclude by discussing how the choice of optimizer is not only about computation—how fast we can get to our solution—but also can have an impact on generalization.[1]

B.1 A Convergence Proof for Gradient Descent

The convergence proof for gradient descent requires only a few assumptions. We will not need the function to be convex; instead, we will only characterize convergence to a stationary point (which might be a local minimum). Instead, we only need two conditions. First, the function should be bounded from below, and so have a non-infinite minimum value $c(\mathbf{w}^*)$. Second, we will need the function to smooth. In particular, we will assume that the gradient of c is Lipschitz continuous, which simply means that there exists an $L > 0$ such that for all $\mathbf{w}, \mathbf{v}$,

$$\|\nabla c(\mathbf{w}) - \nabla c(\mathbf{v})\| \leq L\|\mathbf{w} - \mathbf{v}\| \tag{B.1}$$

where we can use any norm $\|\cdot\|$, but we assume here that it is the ℓ_2 norm. This condition just means that the gradient cannot change arbitrarily fast. It is true for most of our models, including any of our GLMs whether we use linear features or neural networks. The results apply even for ReLU activations, which have some non-differentiable points, because we only need this condition to be true almost everywhere.

It is straightforward to show that Equation (B.1) is equivalent to saying

$$c(\mathbf{w}) \leq c(\mathbf{v}) + \langle \nabla c(\mathbf{v}), \mathbf{w} - \mathbf{v} \rangle + \frac{L}{2}\|\mathbf{w} - \mathbf{v}\|^2 \tag{B.2}$$

The intuition for why this is true is that the right-hand side (rhs) gives us a quadratic approximation around $\mathbf{v}$ that is guaranteed to be above our objective function c, even for $\mathbf{w}$ far away from $\mathbf{v}$. Instead of the standard Taylor series expansion, which uses the Hessian in the second-order term, here we use the more conservative Lipschitz constant, which ensures that second-term is quite large. In fact, practically, it is likely quite a bit too large, and we

[1] This chapter was written using the very nice lecture notes from Mark Schmidt here: https://www.cs.ubc.ca/~schmidtm/Courses/540-W19. Unrelated, Mark—possibly the best optimization person ever—also uses Julia for his Machine Learning course.

could have selected a smaller constant than L and still obtained an upper bound for most $\mathbf{w}$ in a nearby region. But, let's keep this easy on ourselves and use this nice result about Lipschitz functions c.

This property almost immediately gives us the result. Just like when we derived gradient descent, we can minimize this second-order approximation, which tells us to use the stepsize $\eta_t = 1/L$. Then for iterations $\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{L}\nabla c(\mathbf{w}_t)$, we can substitute $\mathbf{w} = \mathbf{w}_{t+1}$ and $\mathbf{v} = \mathbf{w}_t$ in Equation (B.2) to get

$$\begin{aligned}
c(\mathbf{w}_{t+1}) &\leq c(\mathbf{w}_t) + \langle \nabla c(\mathbf{w}_t), \mathbf{w}_{t+1} - \mathbf{w}_t \rangle + \tfrac{L}{2}\|\mathbf{w}_{t+1} - \mathbf{w}_t\|^2 \\
&= c(\mathbf{w}_t) + \langle \nabla c(\mathbf{w}_t), -\tfrac{1}{L}\nabla c(\mathbf{w}_t) \rangle + \tfrac{L}{2}\|\tfrac{1}{L}\nabla c(\mathbf{w}_t)\|^2 && \triangleright\ \mathbf{w}_{t+1} - \mathbf{w}_t = -\frac{1}{L}\nabla c(\mathbf{w}_t) \\
&= c(\mathbf{w}_t) - \tfrac{1}{L}\|\nabla c(\mathbf{w}_t)\|^2 + \tfrac{1}{2L}\|\nabla c(\mathbf{w}_t)\|^2 && \triangleright\ \|\tfrac{1}{L}\nabla c(\mathbf{w}_t)\|^2 = \tfrac{1}{L^2}\|\nabla c(\mathbf{w}_t)\|^2 \\
&= c(\mathbf{w}_t) - \tfrac{1}{2L}\|\nabla c(\mathbf{w}_t)\|^2
\end{aligned}$$

Therefore, on each step we have guaranteed improvement. This improvement slows down as the gradient gets smaller. We can say we have converged when $\|\nabla c(\mathbf{w}_t)\|^2 \leq \epsilon$ for some tolerance $\epsilon > 0$. The guaranteed progress implies we must reach this condition, otherwise the objective value would decrease infinitely, which is not possible because the lowest it can go to is $c(\mathbf{w}^*)$. In the next section, we characterize the rate at which is reaches this point.

B.2 Convergence Rate of Gradient Descent

Notice that

$$c(\mathbf{w}_t) \leq c(\mathbf{w}_{t-1}) - \tfrac{1}{2L}\|\nabla c(\mathbf{w}_{t-1})\|^2 \implies \|\nabla c(\mathbf{w}_{t-1})\|^2 \leq 2L(c(\mathbf{w}_{t-1}) - c(\mathbf{w}_t))$$

and that we have the telescoping sum

$$\begin{aligned}
&\sum_{k=1}^{t}(c(\mathbf{w}_{k-1}) - c(\mathbf{w}_k)) \\
&= c(\mathbf{w}_0) - \underbrace{c(\mathbf{w}_1) + c(\mathbf{w}_1)}_{=0} - \underbrace{c(\mathbf{w}_2) + c(\mathbf{w}_2)}_{=0} - c(\mathbf{w}_3) + \ldots - \underbrace{c(\mathbf{w}_{t-1}) + c(\mathbf{w}_{t-1})}_{=0} - c(\mathbf{w}_t) \\
&= c(\mathbf{w}_0) - c(\mathbf{w}_t)
\end{aligned}$$

Combining these two equations, we get that

$$\sum_{k=1}^{t}\|\nabla c(\mathbf{w}_{k-1})\|^2 \leq 2L\sum_{k=1}^{t}(c(\mathbf{w}_{k-1}) - c(\mathbf{w}_k)) = 2L(c(\mathbf{w}_0) - c(\mathbf{w}_t)) \leq 2L(c(\mathbf{w}_0) - c(\mathbf{w}^*))$$

where the last step follows from the fact that $c(\mathbf{w}_t) \geq c(\mathbf{w}^*)$. Now further for any $k \in \{1, \ldots, t\}$, we know that $\min_{j\in\{1,\ldots,t\}}\|\nabla c(\mathbf{w}_{j-1})\|^2 \leq \|\nabla c(\mathbf{w}_{k-1})\|^2$, by definition of a minimum (the minimum in a set must be smaller than any one of the items in the set). Therefore

$$t \min_{j\in\{1,\ldots,t\}}\|\nabla c(\mathbf{w}_{j-1})\|^2 \leq \sum_{k=1}^{t}\|\nabla c(\mathbf{w}_{k-1})\|^2$$

and so

$$\min_{j\in\{1,\ldots,t\}} \|\nabla c(\mathbf{w}_{j-1})\|^2 \leq \frac{2L}{t}(c(\mathbf{w}_0) - c(\mathbf{w}^*))$$

Therefore, after t iterations we are guaranteed to have seen a gradient less than $1/t$, times a constant that depends on the Lipschitz constant L and how far we started from the optimum $c(\mathbf{w}^*)$. In particular, to guarantee the norm is less than ϵ, we need

$$\frac{2L}{t}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) \leq \epsilon \implies t \geq \frac{2L}{\epsilon}(c(\mathbf{w}_0) - c(\mathbf{w}^*))$$

Therefore, within $t = O(1/\epsilon)$ iterations, we know we will reach our termination condition $\|\nabla c(\mathbf{w}_j)\|^2 \leq \epsilon$ and have converged.

B.3 Convergence Rate of Stochastic Gradient Descent

We can use a similar analysis for mini-batch SGD, by leveraging the fact that we can see SGD as a noisy version of batch GD. For mini-batch $\mathcal{B}_t$ on time step t, we can define

$$\epsilon_t \stackrel{\text{def}}{=} \left(\frac{1}{|\mathcal{B}_t|}\sum_{i\in\mathcal{B}_t} \nabla c_i(\mathbf{w}_t)\right) - \nabla c(\mathbf{w}_t) \tag{B.3}$$

as the stochastic noise introduced from using a mini-batch rather than all the data and rewrite the SGD update as

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t\left(\nabla c(\mathbf{w}_t) + \epsilon_t\right)$$

Again leveraging that c is Lipschitz, with Lipschitz constant L, and using $\mathbf{w}_{t+1} - \mathbf{w}_t = -\eta_t\left(\nabla c(\mathbf{w}_t) + \epsilon_t\right)$, we get that

$$\begin{aligned}
c(\mathbf{w}_{t+1}) &\leq c(\mathbf{w}_t) + \langle \nabla c(\mathbf{w}_t), \mathbf{w}_{t+1} - \mathbf{w}_t\rangle + \tfrac{L}{2}\|\mathbf{w}_{t+1} - \mathbf{w}_t\|^2 \\
&= c(\mathbf{w}_t) + \langle \nabla c(\mathbf{w}_t), -\eta_t\left(\nabla c(\mathbf{w}_t) + \epsilon_t\right)\rangle + \tfrac{L}{2}\|\eta_t\left(\nabla c(\mathbf{w}_t) + \epsilon_t\right)\|^2 \\
&= c(\mathbf{w}_t) - \eta_t\|\nabla c(\mathbf{w}_t)\|^2 - \eta_t\langle \nabla c(\mathbf{w}_t), \epsilon_t\rangle + \tfrac{L\eta_t^2}{2}\|\nabla c(\mathbf{w}_t) + \epsilon_t\|^2
\end{aligned}$$

We want to understand how much progress we make on each step, but now each step is stochastic. Instead, we reason about the expected value of the objective after our update. The bound involves the variance due to this noise from using a mini-batch, which we can assume is upper bounded by a constant $\sigma_t^2 > 0$,

$$\mathbb{E}\left[\|\nabla c(\mathbf{w}_t) + \epsilon_t\|^2\right] \leq \sigma_t^2.$$

Now we can take the expectation over this noise, ϵ_t, that results in $\mathbf{w}_{t+1}$ from the given $\mathbf{w}_t$, and get

$$\begin{aligned}
\mathbb{E}[c(\mathbf{w}_{t+1})] &\leq c(\mathbf{w}_t) - \eta_t\|\nabla c(\mathbf{w}_t)\|^2 - \eta_t\mathbb{E}[\langle \nabla c(\mathbf{w}_t), \epsilon_t\rangle] + \tfrac{L\eta_t^2}{2}\mathbb{E}\left[\|\nabla c(\mathbf{w}_t) + \epsilon_t\|^2\right] \\
&= c(\mathbf{w}_t) - \eta_t\|\nabla c(\mathbf{w}_t)\|^2 - \eta_t\langle \nabla c(\mathbf{w}_t), \underbrace{\mathbb{E}[\epsilon_t]}_{=0}\rangle + \tfrac{L\eta_t^2}{2}\underbrace{\mathbb{E}\left[\|\nabla c(\mathbf{w}_t) + \epsilon_t\|^2\right]}_{\leq\sigma_t^2} \\
&\leq c(\mathbf{w}_t) - \eta_t\|\nabla c(\mathbf{w}_t)\|^2 + \tfrac{L\eta_t^2\sigma_t^2}{2}
\end{aligned}$$

We can go through similar steps to above, now also including expectations and allowing the stepsize η_t to change with each iteration. The final result, with $\eta_{\text{sum}} \stackrel{\text{def}}{=} \sum_{k=0}^{t-1} \eta_k$ and $\eta_{\text{var}} \stackrel{\text{def}}{=} \sum_{k=0}^{t-1} \eta_k^2 \sigma_k^2$, is

$$\min_{j \in \{1,\ldots,t\}} \mathbb{E}\left[\|\nabla c(\mathbf{w}_{j-1})\|^2\right] \leq \frac{1}{\eta_{\text{sum}}}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) + \frac{L}{2}\frac{\eta_{\text{var}}}{\eta_{\text{sum}}} \tag{B.4}$$

This result differs from gradient descent in that we have this additional second term in the upper bound.

We can reason about the values for this second term, depending on different stepsize choices and mini-batch sizes. First, let's assume we have a fixed mini-batch size, and so variability due to the mini-batch is fixed across iterations. In other words, we can assume that σ_t corresponds to a fixed σ for all t. Notice that if the mini-batch size is the whole dataset, then $\sigma^2 = 0$ and so

$$\eta_{\text{var}} = \sum_{k=0}^{t-1} \eta_k^2 \sigma_k^2 = \sigma^2 \sum_{k=0}^{t-1} \eta_k^2 = 0$$

Voila! We get back the same upper bound as gradient descent, because the second term disappears. (This is a good sanity check that the general result applies to our known special case). For mini-batch SGD, though, we expect $\sigma^2 > 0$.

Now let us consider three options for the stepsize: a constant stepsize η, a slowly decreasing stepsize $\eta_t = \eta/\sqrt{t}$ and a quickly decreasing stepsize $\eta_t = \eta/t$. For the constant stepsize we get that

$$\eta_t = \eta \implies \eta_{\text{sum}} = \sum_{k=0}^{t-1} \eta_k = \sum_{k=0}^{t-1} \eta = t\eta \qquad \text{and} \quad \eta_{\text{var}} = \sigma^2 \sum_{k=0}^{t-1} \eta_k^2 = \sigma^2 \sum_{k=0}^{t-1} \eta^2 = t\sigma^2\eta^2$$

Plugging this into the above upper bound in Equation (B.4), we get that

$$\min_{j \in \{0,1,\ldots,t-1\}} \mathbb{E}[\|\nabla c(\mathbf{w}_j)\|^2] \leq \frac{1}{\eta t}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) + \frac{L\sigma^2\eta}{2} \tag{B.5}$$

The first term still converges to zero at the same rate, $O(1/t)$, as gradient descent; this is true even though each update costs much less than a full-batch GD update. How neat! But, there is a cost: the second term prevents us from getting this bound all the way to zero. We reduce error fast in early steps, because we reduce this first term at a rate of $O(1/t)$. Once we start getting near the solution, then we get stuck at the error given by the second term. The constant stepsize helped us get to this region fast, but then we are not robust to the noise in the mini-batch stochastic gradient.

Naturally, we could use a slowly decreasing stepsize, to help be robust to this noise.

$$\eta_t = \eta/\sqrt{t+1} \implies \eta_{\text{sum}} = \sum_{k=0}^{t-1} \eta_k = \eta \sum_{k=0}^{t-1} \tfrac{1}{\sqrt{k+1}} = \eta \sum_{k=1}^{t} \tfrac{1}{\sqrt{k}} \approx \eta\sqrt{t}$$

$$\eta_{\text{var}} = \sigma^2 \sum_{k=0}^{t-1} \eta_k^2 = \sigma^2\eta^2 \sum_{k=1}^{t} \tfrac{1}{k} \approx \sigma^2\eta^2 \log t$$

where the approximations $\sqrt{t}$ and $\log t$ come from well-known formulas for these summations. Plugging this into the above upper bound in Equation (B.4), we get that

$$\min_{j\in\{0,1,\ldots,t-1\}} \mathbb{E}[\|\nabla c(\mathbf{w}_j)\|^2] \leq \frac{1}{\eta\sqrt{t}}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) + \tfrac{L}{2}\sigma^2\eta\tfrac{\log t}{\sqrt{t}} \tag{B.6}$$

Now both terms go to zero at a rate of $O(1/\sqrt{t})$.[2] This is quite a bit slower than gradient descent and SGD with a constant stepsize. For example, after $t = 100$ updates, $\sqrt{t} = 10$ and after $t = 1000$ updates we have that $\sqrt{t} \approx 30$. We can see that $1/t$ gets smaller much faster than $1/\sqrt{t}$. However, now we do have the advantage that the second term goes to zero.

Finally, it is informative to consider what happens if we decrease the stepsize too quickly.

$$\eta_t = \eta/(t+1) \implies \eta_{\text{sum}} = \sum_{k=0}^{t-1}\eta_k = \eta\sum_{k=1}^{t}\tfrac{1}{k} \approx \eta\log t$$
$$\eta_{\text{var}} = \sigma^2\sum_{k=0}^{t-1}\eta_k^2 = \sigma^2\eta^2\sum_{k=1}^{t}\tfrac{1}{k^2} \approx \sigma^2\eta^2$$

giving

$$\min_{j\in\{0,1,\ldots,t-1\}} \mathbb{E}[\|\nabla c(\mathbf{w}_j)\|^2] \leq \frac{1}{\eta\log t}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) + L\sigma^2\eta\tfrac{1}{\log t} \tag{B.7}$$

A rate of $1/\log t$ is a very poor rate. The optimizer stops making real progress too early, and is stuck very very slowly decreasing error. Eventually, it will get there, as $\log t \to \infty$.

The above analysis gives us some insight into how to set our stepsize. It suggests that we should start with more aggressive stepsizes, and then potentially start decaying later in learning. It might be problematic to decay all the way to zero, as we might prevent further learning too quickly. An idea between all of the above is to decay the stepsize to some minimal value—rather than all the way to zero—and consider alternative rules to decide when to start decaying. Many optimization packages out there have options for different stepsize decay schedules, that take into account this understanding of how SGD behaves.

B.4 Selecting the Size of the Mini-batch

The previous sections gives us some insight into selecting the mini-batch size. We see that it is typically not that useful to select the mini-batch size to be the full dataset, because SGD can converge as fast as GD, at least initially. Further, the noise introduced by using a mini-batch actually seems to improve generalization performance, through implicit regularization. However, picking a mini-batch of size $b = 1$ is rarely (if ever) as effective as picking $b > 1$. So what is the right choice, between $b = 1$ and $b = n$?

Let's assume that we have a fixed stepsize η. Recall that our upper bound, giving us our convergence rate, was $\frac{1}{\eta t}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) + \frac{L\sigma^2\eta}{2}$. This upper bound is smaller for bigger η and smaller σ^2. The σ^2 is smaller for bigger b. In fact, the implicit regularization results show that the weight on the implicit regularizer is approximately proportional to η/b, for reasonably small b (number of samples n much bigger than b, written $n \gg b$). This result

[2] It is more correct to say the rate is $O(\log t/\sqrt{t})$, but $\log t$ grows so slowly that the primary factor is the denominator with $\sqrt{t}$.

also suggests that if we increase our stepsize, then it is reasonable to increase the batch-size to maintain the same level of implicit regularization, again up to a point where the ratio of the batch-size to number of samples is small. For example, if we have $n = 1$ million, then we likely would not pick b to be any larger than 2048. This perspective suggests that we should pick the largest, reasonably small mini-batch size (e.g., 2048).

There is yet one more option to consider, which is computation. If we can perfectly parallelize computation of the mini-batch, then it is a no-brainer to pick a larger mini-batch size. However, what if the number of gradients we can compute in parallel, m, is less that this mini-batch size? (Say our compute can only do 32). Now should we opt to use a mini-batch of size $b = m$ and do more updates, or pick $b > m$ and do fewer updates?

We can try to reason about this using our bound again. Let's imagine, that if we could, we would pick an ideal batch size of b^* (say $b^* = 2048$, the largest reasonably small mini-batch size) with correspond ideal learning rate η^*. But, we can only compute $m < b^*$ gradients in parallel. Let's contrast to using $b = m$. If we use the heuristic above, the ratio between the stepsize and mini-batch size should stay the same, so we need to pick η such that $\eta/m = \eta^*/b^*$, giving $\eta = \frac{m}{b^*}\eta^*$. This stepsize is smaller, because our mini-batch is smaller. For each iteration using b^* mini-batches, we have to take b^*/m computation steps. Therefore, if we did t^* iterations using b^* mini-batches, we would have been able to do more iterations $t = \frac{b^*}{m}t^*$ only using mini-batches of size m. Plugging this into our bound, we get

$$\begin{aligned}\frac{1}{\eta t}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) + \frac{L\sigma^2\eta}{2} &= \frac{1}{\frac{m}{b^*}\eta^*\frac{b^*}{m}t^*}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) + \frac{L\sigma^2\frac{m}{b^*}\eta^*}{2} \\ &= \frac{1}{\eta^* t^*}(c(\mathbf{w}_0) - c(\mathbf{w}^*)) + \frac{L\sigma^2\frac{m}{b^*}\eta^*}{2}\end{aligned}$$

The first term is actually the same for both! And the second term is approximately the same for both too. To understand why, recall that the variance of the mini-batch is proportional to b, meaning $\sigma^2 \propto 1/b$. Therefore, the variance σ_*^2 for b^* is proportional to $1/b^*$ and so the ratio $\sigma^2\eta/(\sigma_*^2\eta^*) = (1/m)\frac{m}{b^*}\eta^*/((1/b^*)\eta^*) = 1$. The conclusion from this is that it does not matter which we pick: we can pick our mini-batch to be size $b = m$ and do more updates or choose to do fewer updates and estimate up to a mini-batch of size b^*. (But we still keep the mini-batches reasonably small, since $b^* << n$.) Under this outcome, it might actually make sense to pick $b = m$, to ensure that we do not have to pick too big of a stepsize to get the nice regularization properties of SGD.

This reasoning, of course, is using our upper bound rather than a true convergence rate, so it is not perfect. Interestingly, though, there is a comprehensive empirical study that corroborates that this linear relationship holds, at least for smaller mini-batch sizes [25]. This suggests that a reasonably safe option is to set $b = m$, since there is definitely a convergence rate improvement that is linear in increasing batch-size, when going from $b = 1$ to smaller mini-batch sizes. At some point, there is diminishing returns where it was wasteful to increase the mini-batch size further. Once we fix b, we do still have to tune our stepsize, but at least now we only have one hyperparameter to tune.

Appendix C

Exercise Solutions

C.1 Chapter 2 Exercises

Solution to Exercise 7

$$\nabla_{\boldsymbol{\Sigma}} - \ln p(\mathbf{x}) = \nabla_{\boldsymbol{\Sigma}} - \ln(2\pi)^{-d/2} + \nabla_{\boldsymbol{\Sigma}} \tfrac{1}{2} \ln |\boldsymbol{\Sigma}| + \nabla_{\boldsymbol{\Sigma}} \frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \tag{C.1}$$

As before the first term disappears. However, the second two terms both include $\boldsymbol{\Sigma}$. The gradient of the log of the determinant of this matrix is $\boldsymbol{\Sigma}^{-1}$ (see [21, Equation 48]). The same resource gives us an easy rule to compute the gradient of the second term, without having to do so ourselves using partial derivatives. Namely, we know that for a symmetric matrix $\mathbf{X}$ (see [21, Equation 52])

$$\nabla_{\mathbf{X}} \mathbf{a}^\top \mathbf{X}^{-1} \mathbf{a} = - \mathbf{X}^{-1} \mathbf{a} \mathbf{a}^\top \mathbf{X}^{-1}$$

where we use the fact that $\mathbf{X}^\top = \mathbf{X}$ for a symmetric matrix. Therefore, we can plug these into Equation (C.1) to get that

$$\nabla_{\boldsymbol{\Sigma}} - \ln p(\mathbf{x}) = \tfrac{1}{2} \boldsymbol{\Sigma}^{-1} - \frac{1}{2} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}$$

Now we can use this to find

$$\begin{aligned}
&\nabla_{\boldsymbol{\Sigma}} - \sum_{i=1}^{n} \ln p(\mathbf{x}_i) = \tfrac{1}{2} n \boldsymbol{\Sigma}^{-1} - \frac{1}{2} \sum_{i=1}^{n} \boldsymbol{\Sigma}^{-1} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} = \mathbf{0} \\
&\implies n \boldsymbol{\Sigma}^{-1} = \boldsymbol{\Sigma}^{-1} \left[\sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\top \right] \boldsymbol{\Sigma}^{-1} \\
&\implies n \mathbf{I} = \left[\sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\top \right] \boldsymbol{\Sigma}^{-1} \qquad \triangleright \text{left multiply both sides by } \boldsymbol{\Sigma} \\
&\implies n \boldsymbol{\Sigma} = \sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\top \qquad \triangleright \text{right multiply both sides by } \boldsymbol{\Sigma} \\
&\implies \boldsymbol{\Sigma} = \tfrac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\top
\end{aligned}$$

C.2 Chapter 3 Exercises

Solution to Exercise 11

$$\begin{aligned}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 &= (\mathbf{X}\mathbf{w}-\mathbf{y})^\top(\mathbf{X}\mathbf{w}-\mathbf{y}) \\ &= \sum_{i=1}^{n}(\mathbf{X}_{i:}\mathbf{w}-y_i)^2 = \sum_{i=1}^{n}(\mathbf{x}_i\mathbf{w}-y_i)^2.\end{aligned}$$

To compute this gradient, we can use the standard approach from before, with partial derivatives. However, for these relatively simple objectives, it is actually simpler to use the basic rules for computing gradient with matrices and vector, shown in Table 1.1. First, notice that

$$\begin{aligned}(\mathbf{X}\mathbf{w}-\mathbf{y})^\top(\mathbf{X}\mathbf{w}-\mathbf{y}) &= (\mathbf{w}^\top\mathbf{X}^\top-\mathbf{y}^\top)(\mathbf{X}\mathbf{w}-\mathbf{y}) && \text{because } (\mathbf{A}\mathbf{B})^\top=\mathbf{B}^\top\mathbf{A}^\top \\ &= \mathbf{w}^\top\mathbf{X}^\top\mathbf{X}\mathbf{w}-2\mathbf{y}^\top\mathbf{X}\mathbf{w}+\mathbf{y}^\top\mathbf{y} && \text{because } \mathbf{y}^\top\mathbf{X}\mathbf{w}=\mathbf{w}^\top\mathbf{X}^\top\mathbf{y}\end{aligned}$$

Using the derivative rules for vectors and matrices, where $\nabla\mathbf{w}^\top\mathbf{A}\mathbf{w}=2\mathbf{A}\mathbf{w}$ and $\nabla\mathbf{b}^\top\mathbf{A}\mathbf{w}=\mathbf{A}^\top\mathbf{b}$,[1] we get

$$\nabla\|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 = 2\mathbf{X}^\top\mathbf{X}\mathbf{w}-2\mathbf{X}^\top\mathbf{y}$$

C.3 Representation Exercises

Solution to Exercise 29

$$\begin{aligned}\frac{\partial c(\mathbf{W}^{(1)},\mathbf{W}^{(2)})}{\partial\mathbf{W}_j^{(1)}} &= \frac{\partial\ell(\hat{y},y)}{\partial\mathbf{W}_j^{(1)}} \\ &= \left(\frac{\partial\ell(\hat{y},y)}{\partial\hat{y}}\right)\frac{\partial\hat{y}}{\partial\mathbf{W}_j^{(1)}} \\ &= \left(\frac{\partial\ell(\hat{y},y)}{\partial\hat{y}}\right)\frac{\partial\sigma(\boldsymbol{\theta}^{(1)})}{\partial\boldsymbol{\theta}^{(1)}}\frac{\partial\boldsymbol{\theta}^{(1)}}{\partial\mathbf{W}_j^{(1)}} && \triangleright\ \boldsymbol{\theta}^{(1)}=\mathbf{h}^{(1)}\mathbf{W}^{(1)} \\ &= \left(\frac{\partial\ell(\hat{y},y)}{\partial\hat{y}}\right)\frac{\partial\sigma(\boldsymbol{\theta}^{(1)})}{\partial\boldsymbol{\theta}^{(1)}}\mathbf{h}_j && \triangleright\ \frac{\partial\boldsymbol{\theta}^{(1)}}{\partial\mathbf{W}_j^{(1)}}=\frac{\mathbf{h}^{(1)}\mathbf{W}^{(1)}}{\partial\mathbf{W}_j^{(1)}}=\frac{\sum_{i=1}^{p_1}\mathbf{h}_i^{(1)}\mathbf{W}_i^{(1)}}{\partial\mathbf{W}_j^{(1)}}=\mathbf{h}_j^{(1)}\end{aligned}$$

Now we simply need to compute

$$\begin{aligned}\frac{\partial\ell(\hat{y},y)}{\partial\hat{y}} &= \frac{\partial -y\ln\hat{y}-(1-y)\ln(1-\hat{y})}{\partial\hat{y}} \\ &= -y\frac{\partial\ln\hat{y}}{\partial\hat{y}}-(1-y)\frac{\partial\ln(1-\hat{y})}{\partial\hat{y}} \\ &= -y\frac{1}{\hat{y}}-(1-y)\frac{-1}{1-\hat{y}}\end{aligned}$$

[1] Notice if $\mathbf{A}$ is a scalar and $\mathbf{w}$ a scalar, this result is intuitive: the derivative of $w\cdot a\cdot w$ is the derivative of aw^2, which is $2aw$.