Chapter 9

Regularization and model selection

9.1 Regularization

Recall that as discussed in Section 8.1, overftting is typically a result of using too complex models, and we need to choose a proper model complexity to achieve the optimal bias-variance tradeoff. When the model complexity is measured by the number of parameters, we can vary the size of the model (e.g., the width of a neural net). However, the correct, informative complexity measure of the models can be a function of the parameters (e.g., ℓ_2 norm of the parameters), which may not necessarily depend on the number of parameters. In such cases, we will use regularization, an important technique in machine learning, control the model complexity and prevent overfitting.

Regularization typically involves adding an additional term, called a regularizer and denoted by $R(\theta)$ here, to the training loss/cost function:

$$J_\lambda(\theta) = J(\theta) + \lambda R(\theta) \tag{9.1}$$

Here J_λ is often called the regularized loss, and $\lambda \geq 0$ is called the regularization parameter. The regularizer $R(\theta)$ is a nonnegative function (in almost all cases). In classical methods, $R(\theta)$ is purely a function of the parameter θ, but some modern approach allows $R(\theta)$ to depend on the training dataset.[1]

The regularizer $R(\theta)$ is typically chosen to be some measure of the complexity of the model θ. Thus, when using the regularized loss, we aim to find a model that both fit the data (a small loss $J(\theta)$) and have a small

[1]Here our notations generally omit the dependency on the training dataset for simplicity—we write $J(\theta)$ even though it obviously needs to depend on the training dataset.

model complexity (a small $R(\theta)$). The balance between the two objectives is controlled by the regularization parameter λ. When $\lambda = 0$, the regularized loss is equivalent to the original loss. When λ is a sufficiently small positive number, minimizing the regularized loss is effectively minimizing the original loss with the regularizer as the tie-breaker. When the regularizer is extremely large, then the original loss is not effective (and likely the model will have a large bias.)

The most commonly used regularization is perhaps ℓ_2 regularization, where $R(\theta) = \frac{1}{2}\|\theta\|_2^2$. It encourages the optimizer to find a model with small ℓ_2 norm. In deep learning, it's oftentimes referred to as **weight decay**, because gradient descent with learning rate η on the regularized loss $R_\lambda(\theta)$ is equivalent to shrinking/decaying θ by a scalar factor of $1 - \eta\lambda$ and then applying the standard gradient

$$\begin{aligned} \theta &\leftarrow \theta - \eta\nabla J_\lambda(\theta) = \theta - \eta\lambda\theta - \eta\nabla J(\theta) \\ &= \underbrace{(1-\lambda\eta)\theta}_{\text{decaying weights}} - \eta\nabla J(\theta) \end{aligned} \tag{9.2}$$

Besides encouraging simpler models, regularization can also impose inductive biases or structures on the model parameters. For example, suppose we had a prior belief that the number of non-zeros in the ground-truth model parameters is small,[2]—which is oftentimes called sparsity of the model—, we can impose a regularization on the number of non-zeros in θ, denoted by $\|\theta\|_0$, to leverage such a prior belief. Imposing additional structure of the parameters narrows our search space and makes the complexity of the model family smaller,—e.g., the family of sparse models can be thought of as having lower complexity than the family of all models—, and thus tends to lead to a better generalization. On the other hand, imposing additional structure may risk increasing the bias. For example, if we regularize the sparsity strongly but no sparse models can predict the label accurately, we will suffer from large bias (analogously to the situation when we use linear models to learn data than can only be represented by quadratic functions in Section 8.1.)

The sparsity of the parameters is not a continuous function of the parameters, and thus we cannot optimize it with (stochastic) gradient descent. A common relaxation is to use $R(\theta) = \|\theta\|_1$ as a continuous surrogate.[3]

[2]For linear models, this means the model just uses a few coordinates of the inputs to make an accurate prediction.

[3]There has been a rich line of theoretical work that explains why $\|\theta\|_1$ is a good surrogate for encouraging sparsity, but it's beyond the scope of this course. An intuition is: assuming the parameter is on the unit sphere, the parameter with smallest ℓ_1 norm also

The $R(\theta) = \|\theta\|_1$ (also called LASSO) and $R(\theta) = \frac{1}{2}\|\theta\|_2^2$ are perhaps among the most commonly used regularizers for linear models. Other norm and powers of norms are sometimes also used. The ℓ_2 norm regularization is much more commonly used with kernel methods because ℓ_1 regularization is typically not compatible with the kernel trick (the optimal solution cannot be written as functions of inner products of features.)

In deep learning, the most commonly used regularizer is ℓ_2 regularization or weight decay. Other common ones include dropout, data augmentation, regularizing the spectral norm of the weight matrices, and regularizing the Lipschitzness of the model, etc. Regularization in deep learning is an active research area, and it's known that there is another implicit source of regularization, as discussed in the next section.

9.2 Implicit regularization effect (optional reading)

The implicit regularization effect of optimizers, or implicit bias or algorithmic regularization, is a new concept/phenomenon observed in the deep learning era. It largely refers to that the optimizers can implicitly impose structures on parameters beyond what has been imposed by the regularized loss.

In most classical settings, the loss or regularized loss has a unique global minimum, and thus any reasonable optimizer should converge to that global minimum and cannot impose any additional preferences. However, in deep learning, oftentimes the loss or regularized loss has more than one (approximate) global minima, and difference optimizers may converge to different global minima. Though these global minima have the same or similar training losses, they may be of different nature and have dramatically different generalization performance. See Figures 9.1 and 9.2 and its caption for an illustration and some experiment results. For example, it's possible that one global minimum gives a much more Lipschitz or sparse model than others and thus has a better test error. It turns out that many commonly-used optimizers (or their components) prefer or bias towards finding global minima of certain properties, leading to a better test performance.

happen to be the sparsest parameter with only 1 non-zero coordinate. Thus, sparsity and ℓ_1 norm gives the same extremal points to some extent.

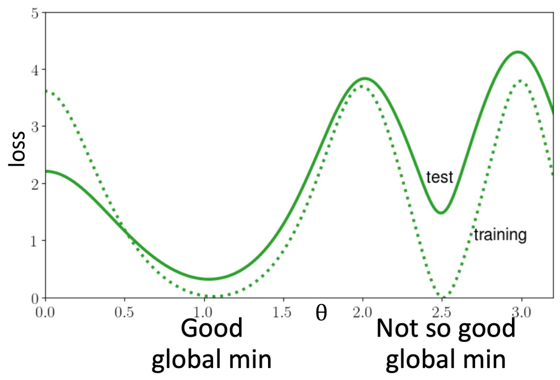
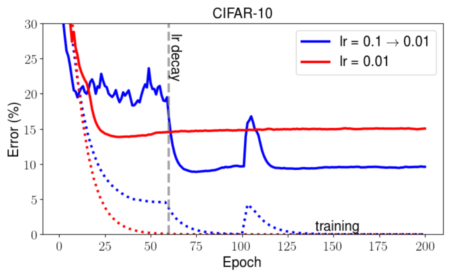


Figure 9.1: An Illustration that different global minima of the training loss can have different test performance.



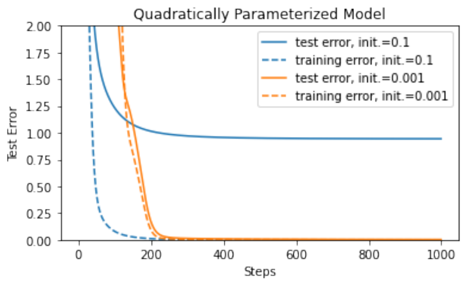


Figure 9.2: **Left:** Performance of neural networks trained by two different learning rates schedules on the CIFAR-10 dataset. Although both experiments used exactly the same regularized losses and the optimizers fit the training data perfectly, the models' generalization performance differ much. **Right:** On a different synthetic dataset, optimizers with different initializations have the same training error but different generalization performance.[4]

In summary, the takehome message here is that the choice of optimizer does not only affect minimizing the training loss, but also imposes implicit regularization and affects the generalization of the model. Even if your current optimizer already converges to a small training error perfectly, you may still need to tune your optimizer for a better generalization, .

[4]The setting is the same as in Woodworth et al. [2020], HaoChen et al. [2020]

One may wonder which components of the optimizers bias towards what type of global minima and what type of global minima may generalize better. These are open questions that researchers are actively investigating. Empirical and theoretical research have offered some clues and heuristics. In many (but definitely far from all) situations, among those setting where optimization can succeed in minimizing the training loss, the use of larger initial learning rate, smaller initialization, smaller batch size, and momentum appears to help with biasing towards more generalizable solutions. A conjecture (that can be proven in certain simplified case) is that stochasticity in the optimization process help the optimizer to find flatter global minima (global minima where the curvature of the loss is small), and flat global minima tend to give more Lipschitz models and better generalization. Characterizing the implicit regularization effect formally is still a challenging open research question.

9.3 Model selection via cross validation

Suppose we are trying select among several different models for a learning problem. For instance, we might be using a polynomial regression model $h_\theta(x) = g(\theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_k x^k)$, and wish to decide if k should be 0, 1, . . . , or 10. How can we automatically select a model that represents a good tradeoff between the twin evils of bias and variance[5]? Alternatively, suppose we want to automatically choose the bandwidth parameter τ for locally weighted regression, or the parameter C for our ℓ_1-regularized SVM. How can we do that?

For the sake of concreteness, in these notes we assume we have some finite set of models $\mathcal{M} = \{M_1, \ldots, M_d\}$ that we're trying to select among. For instance, in our first example above, the model M_i would be an i-th degree polynomial regression model. (The generalization to infinite $\mathcal{M}$ is not hard.[6]) Alternatively, if we are trying to decide between using an SVM, a neural network or logistic regression, then $\mathcal{M}$ may contain these models.

[5]Given that we said in the previous set of notes that bias and variance are two very different beasts, some readers may be wondering if we should be calling them "twin" evils here. Perhaps it'd be better to think of them as non-identical twins. The phrase "the fraternal twin evils of bias and variance" doesn't have the same ring to it, though.

[6]If we are trying to choose from an infinite set of models, say corresponding to the possible values of the bandwidth $\tau \in \mathbb{R}^+$, we may discretize τ and consider only a finite number of possible values for it. More generally, most of the algorithms described here can all be viewed as performing optimization search in the space of models, and we can perform this search over infinite model classes as well.

Cross validation. Lets suppose we are, as usual, given a training set S. Given what we know about empirical risk minimization, here's what might initially seem like a algorithm, resulting from using empirical risk minimization for model selection:

1. Train each model M_i on S, to get some hypothesis h_i.
2. Pick the hypotheses with the smallest training error.

This algorithm does *not* work. Consider choosing the degree of a polynomial. The higher the degree of the polynomial, the better it will fit the training set S, and thus the lower the training error. Hence, this method will always select a high-variance, high-degree polynomial model, which we saw previously is often poor choice.

Here's an algorithm that works better. In **hold-out cross validation** (also called **simple cross validation**), we do the following:

1. Randomly split S into S_{train} (say, 70% of the data) and S_{cv} (the remaining 30%). Here, S_{cv} is called the hold-out cross validation set.
2. Train each model M_i on S_{train} only, to get some hypothesis h_i.
3. Select and output the hypothesis h_i that had the smallest error $\hat{\varepsilon}_{S_{\text{cv}}}(h_i)$ on the hold out cross validation set. (Here $\hat{\varepsilon}_{S_{\text{cv}}}(h)$ denotes the average error of h on the set of examples in S_{cv}.) The error on the hold out validation set is also referred to as the validation error.

By testing/validating on a set of examples S_{cv} that the models were not trained on, we obtain a better estimate of each hypothesis h_i's true generalization/test error. Thus, this approach is essentially picking the model with the smallest estimated generalization/test error. The size of the validation set depends on the total number of available examples. Usually, somewhere between $1/4 - 1/3$ of the data is used in the hold out cross validation set, and 30% is a typical choice. However, when the total dataset is huge, validation set can be a smaller fraction of the total examples as long as the absolute number of validation examples is decent. For example, for the ImageNet dataset that has about 1M training images, the validation set is sometimes set to be 50K images, which is only about 5% of the total examples.

Optionally, step 3 in the algorithm may also be replaced with selecting the model M_i according to $\arg\min_i \hat{\varepsilon}_{S_{\text{cv}}}(h_i)$, and then retraining M_i on the entire training set S. (This is often a good idea, with one exception being learning algorithms that are be very sensitive to perturbations of the initial conditions and/or data. For these methods, M_i doing well on S_{train} does not necessarily mean it will also do well on S_{cv}, and it might be better to forgo this retraining step.)

The disadvantage of using hold out cross validation is that it "wastes" about 30% of the data. Even if we were to take the optional step of retraining the model on the entire training set, it's still as if we're trying to find a good model for a learning problem in which we had $0.7n$ training examples, rather than n training examples, since we're testing models that were trained on only $0.7n$ examples each time. While this is fine if data is abundant and/or cheap, in learning problems in which data is scarce (consider a problem with $n = 20$, say), we'd like to do something better.

Here is a method, called **k-fold cross validation**, that holds out less data each time:

1. Randomly split S into k disjoint subsets of m/k training examples each. Lets call these subsets $S_1, \ldots, S_k$.

2. For each model M_i, we evaluate it as follows:

   For $j = 1, \ldots, k$

   Train the model M_i on $S_1 \cup \cdots \cup S_{j-1} \cup S_{j+1} \cup \cdots S_k$ (i.e., train on all the data except S_j) to get some hypothesis h_{ij}.

   Test the hypothesis h_{ij} on S_j, to get $\hat{\varepsilon}_{S_j}(h_{ij})$.

   The estimated generalization error of model M_i is then calculated as the average of the $\hat{\varepsilon}_{S_j}(h_{ij})$'s (averaged over j).

3. Pick the model M_i with the lowest estimated generalization error, and retrain that model on the entire training set S. The resulting hypothesis is then output as our final answer.

A typical choice for the number of folds to use here would be $k = 10$. While the fraction of data held out each time is now $1/k$—much smaller than before—this procedure may also be more computationally expensive than hold-out cross validation, since we now need train to each model k times.

While $k = 10$ is a commonly used choice, in problems in which data is really scarce, sometimes we will use the extreme choice of $k = m$ in order to leave out as little data as possible each time. In this setting, we would repeatedly train on all but one of the training examples in S, and test on that held-out example. The resulting $m = k$ errors are then averaged together to obtain our estimate of the generalization error of a model. This method has its own name; since we're holding out one training example at a time, this method is called **leave-one-out cross validation.**

Finally, even though we have described the different versions of cross validation as methods for selecting a model, they can also be used more simply to evaluate a *single* model or algorithm. For example, if you have implemented some learning algorithm and want to estimate how well it performs for your application (or if you have invented a novel learning algorithm and want to report in a technical paper how well it performs on various test sets), cross validation would give a reasonable way of doing so.

9.4 Bayesian statistics and regularization

In this section, we will talk about one more tool in our arsenal for our battle against overfitting.

At the beginning of the quarter, we talked about parameter fitting using maximum likelihood estimation (MLE), and chose our parameters according to

$$\theta_{\mathrm{MLE}} = \arg\max_{\theta} \prod_{i=1}^{n} p(y^{(i)}|x^{(i)};\theta).$$

Throughout our subsequent discussions, we viewed θ as an unknown parameter of the world. This view of the θ as being *constant-valued but unknown* is taken in **frequentist** statistics. In the frequentist this view of the world, θ is not random—it just happens to be unknown—and it's our job to come up with statistical procedures (such as maximum likelihood) to try to estimate this parameter.

An alternative way to approach our parameter estimation problems is to take the **Bayesian** view of the world, and think of θ as being a *random variable* whose value is unknown. In this approach, we would specify a **prior distribution** $p(\theta)$ on θ that expresses our "prior beliefs" about the parameters. Given a training set $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$, when we are asked to make a prediction on a new value of x, we can then compute the posterior distribution on the parameters

$$\begin{aligned} p(\theta|S) &= \frac{p(S|\theta)p(\theta)}{p(S)} \\ &= \frac{\left(\prod_{i=1}^{n} p(y^{(i)}|x^{(i)},\theta)\right) p(\theta)}{\int_{\theta} \left(\prod_{i=1}^{n} p(y^{(i)}|x^{(i)},\theta)p(\theta)\right) d\theta} \end{aligned} \tag{9.3}$$

In the equation above, $p(y^{(i)}|x^{(i)},\theta)$ comes from whatever model you're using for your learning problem. For example, if you are using Bayesian logistic regression, then you might choose $p(y^{(i)}|x^{(i)}, \theta) = h_\theta(x^{(i)})^{y^{(i)}}(1-h_\theta(x^{(i)}))^{(1-y^{(i)})}$, where $h_\theta(x^{(i)}) = 1/(1+\exp(-\theta^T x^{(i)}))$.[7]

When we are given a new test example x and asked to make it prediction on it, we can compute our posterior distribution on the class label using the posterior distribution on θ:

$$p(y|x, S) = \int_\theta p(y|x, \theta)p(\theta|S)d\theta \tag{9.4}$$

In the equation above, $p(\theta|S)$ comes from Equation (9.3). Thus, for example, if the goal is to the predict the expected value of y given x, then we would output[8]

$$\mathrm{E}[y|x, S] = \int_y y p(y|x, S)dy$$

The procedure that we've outlined here can be thought of as doing "fully Bayesian" prediction, where our prediction is computed by taking an average with respect to the posterior $p(\theta|S)$ over θ. Unfortunately, in general it is computationally very difficult to compute this posterior distribution. This is because it requires taking integrals over the (usually high-dimensional) θ as in Equation (9.3), and this typically cannot be done in closed-form.

Thus, in practice we will instead approximate the posterior distribution for θ. One common approximation is to replace our posterior distribution for θ (as in Equation 9.4) with a single point estimate. The **MAP (maximum a posteriori)** estimate for θ is given by

$$\theta_{\mathrm{MAP}} = \arg\max_\theta \prod_{i=1}^{n} p(y^{(i)}|x^{(i)}, \theta)p(\theta). \tag{9.5}$$

Note that this is the same formulas as for the MLE (maximum likelihood) estimate for θ, except for the prior $p(\theta)$ term at the end.

In practical applications, a common choice for the prior $p(\theta)$ is to assume that $\theta \sim \mathcal{N}(0, \tau^2 I)$. Using this choice of prior, the fitted parameters θ_{MAP} will have smaller norm than that selected by maximum likelihood. In practice, this causes the Bayesian MAP estimate to be less susceptible to overfitting than the ML estimate of the parameters. For example, Bayesian logistic regression turns out to be an effective algorithm for text classification, even though in text classification we usually have $d \gg n$.

[7]Since we are now viewing θ as a random variable, it is okay to condition on it value, and write "$p(y|x, \theta)$" instead of "$p(y|x; \theta)$."

[8]The integral below would be replaced by a summation if y is discrete-valued.