Lecture 5: Generalization

CSC311, Fall 2020

Based on notes by Roger Grosse

1 Introduction

When we train a machine learning model, we don't just want it to learn to model the training data. We want it to *generalize* to data it hasn't seen before. Fortunately, there's a convenient way to measure an algorithm's generalization performance: we measure its performance on a held-out test set, consisting of examples it hasn't seen before. If an algorithm works well on the training set but fails to generalize, we say it is *overfitting*. This note will cover the basics of generalization including validation, the bias variance tradeoff, and a review of the regularization strategies we've seen so far.

1.1 Learning Goals

- Know the difference between training, validation, and test sets.
- Be able to reason qualitatively about how training and test error depend on the size of the model, the number of training examples, and the number of training iterations.
- Decompose generalization error into bias, variance, and Bayes error.
- Review several strategies to improve generalization: reducing model capacity, L2 regularization / weight decay, early stopping, ensembles

2 Measuring generalization

So far in this course, we've trained our models by optimizing a cost function that is defined as the average loss over the *training* set:

$$\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x}^{(i)}), t^{(i)}). \tag{1}$$

But we don't just want to get the training examples right; we want our models to generalize to novel instances they haven't seen before. To estimate generalization performance, we can partition our data into three subsets:

- A **training set**, a set of training examples the model is trained on.
- A **validation set**, which is used to tune hyperparameters such as the number of hidden units, or the learning rate.
- A **test set**, which is used to measure the generalization performance.

We saw a variation on this basic strategy called cross-validation, in Homework 1, which is often used in situations with small datasets, i.e. less than a few thousand examples.

The losses on these subsets are called **training, validation, and test loss**, respectively. Hopefully it's clear why we need separate training and test sets: if we train on the test data, we have no idea whether the model is correctly generalizing, or whether it's simply memorizing the training examples. It's a more subtle point why we need a separate validation set.

- We can't tune hyperparameters on the training set, because we want to choose values that will generalize. For instance, if we choose a very expressive model that easily memorizes the training data, it will generalize poorly. Tuning on the training data could lead us to choose such a model.
- We also can't tune them on the test set, because that would be "cheating." We're only allowed to use the test set once, to report the final performance. If we "peek" at the test data by using it to tune hyperparameters, it will no longer give a realistic estimate of generalization performance.

The most basic strategy for tuning hyperparameters is to do a **grid search**: for each hyperparameter, choose a set of candidate values. Separately train models using all possible combinations of these values, and choose whichever configuration gives the best validation error. A closely related alternative is **random search**: train a bunch of models using random configurations of the hyperparameters, and pick whichever one has the best validation error. The advantage of random search over grid search is as follows: suppose your model has 10 hyperparameters, but only two of them are actually important. (You don't know which two.) It's infeasible to do a grid search in 10 dimensions, but random search still ought to provide reasonable coverage of the 2-dimensional space of the important hyperparameters. On the other hand, in a scientific setting, grid search has the advantage that it's easy to reproduce the exact experimental setup.

3 Reasoning about generalization

If a model performs well on the training set but generalizes badly, we say it is **overfitting**. A model might overfit if the training set contains **accidental regularities**. For instance, if the task is to classify handwritten digits, it might happen that in the training set, all images of 9's have pixel number 122 on, while all other examples have it off. The model might learn to exploit this accidental regularity, thereby correctly classifying all the training examples of 9's, without learning the true regularities. If this property doesn't hold on the test set, the model will generalize badly.

As an extreme case, remember the neural network we constructed in Lecture 4, which was able to learn arbitrary Boolean functions? It has a separate hidden unit for every possible input configuration. This model is able to **memorize** a training set, i.e. learn the correct answer for every training example, even though it will have no idea how to classify novel instances. The problem is that this model has too large a **capacity**, i.e. ability to remember information about its training data. Capacity isn't a formal term, but corresponds roughly to the number of trainable parameters.

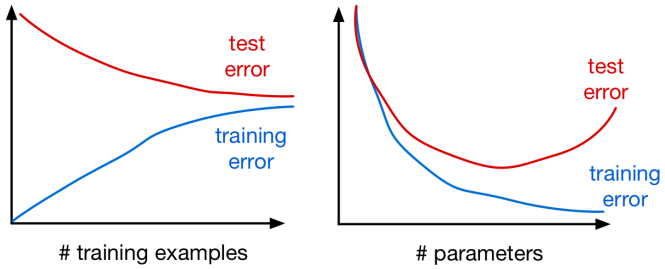

Figure 1: **(left)** Qualitative relationship between the number of training examples and training/test error. **(right)** Qualitative relationship between the number of parameters (or model capacity) and training/test error.

In order to reason qualitatively about generalization, let's think about how the training and generalization error vary as a function of the number of training examples and the number of parameters. Having more training data should only help generalization: for any particular test example, the larger the training set, the more likely there will be a closely related training example. Also, the larger the training set, the fewer the accidental regularities, so the model will be forced to pick up the true regularities. Therefore, generalization error ought to improve as we add more training examples. On the other hand, small training sets are easier to memorize than large ones, so training error tends to increase as we add more examples. As the training set gets larger, the two will eventually meet. This is shown qualitatively in Figure 1.

If the test error *increases* with the number of training examples, that's a sign that you have a bug in your code or that there's something wrong with your model.

Now let's think about the model capacity. As we add more parameters, it becomes easier to fit both the accidental and the true regularities of the training data. Therefore, training error improves as we add more parameters. The effect on generalization error is a bit more subtle. If the model has too little capacity, it generalizes badly because it fails to pick up the regularities (true or accidental) in the data (we say that it is **underfitting**). If it has too much capacity, it will memorize the training set and fail to generalize. Therefore, the effect of capacity on test error is non-monotonic: it decreases, and then increases. We would like to pick models that have enough capacity to learn the true regularities in the training data, but not enough capacity to simply memorize the training set or exploit accidental regularities. This is shown qualitatively in Figure 1.

3.1 Bias and variance

For now, let's focus on squared error loss. We'd like to mathematically model the generalization error. To formalize this, we need to introduce the **data generating distribution**, a hypothetical distribution $p_{\mathcal{D}}(\mathbf{x}, t)$ that all the training and test data are assumed to have come from. We don't need to assume anything about the form of the distribution, so the only nontrivial assumption we're making here is that the training and test data are drawn from the same distribution.

Suppose we have a test input $\mathbf{x}$, and we make a prediction y (which,

for now, we treat as arbitrary). We're interested in the expected error if the targets are sampled from the conditional distribution $p_{\mathcal{D}}(t \mid \mathbf{x})$. By applying the properties of expectation and variance, we can decompose this expectation into two terms:

This derivation makes use of the formula $\mathrm{Var}[z] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$ for a random variable z.

$$\begin{aligned}
\mathbb{E}[(y-t)^2 \mid \mathbf{x}] &= \mathbb{E}[y^2 - 2yt + t^2 \mid \mathbf{x}] \\
&= y^2 - 2y\mathbb{E}[t \mid \mathbf{x}] + \mathbb{E}[t^2 \mid \mathbf{x}] && \text{by linearity of expectation} \\
&= y^2 - 2y\mathbb{E}[t \mid \mathbf{x}] + \mathbb{E}[t \mid \mathbf{x}]^2 + \mathrm{Var}[t \mid \mathbf{x}] && \text{by the formula for variance} \\
&= (y - \mathbb{E}[t \mid \mathbf{x}])^2 + \mathrm{Var}[t \mid \mathbf{x}] \\
&\triangleq (y - y_\star)^2 + \mathrm{Var}[t \mid \mathbf{x}],
\end{aligned}$$

where in the last step we introduce $y_\star = \mathbb{E}[t \mid \mathbf{x}]$, which is the best possible prediction we can make, because the first term is nonnegative and the second term doesn't depend on y. The second term is known as the **Bayes error**, and corresponds to the best possible generalization error we can achieve even if we model the data perfectly.

Now let's treat y as a random variable. Assume we repeat the following experiment: sample a training set randomly from $p_{\mathcal{D}}$, train our model, and compute its predictions on $\mathbf{x}$. If we suppress the dependence on $\mathbf{x}$ for simplicity, the expected squared error decomposes as:

$$\begin{aligned}
\mathbb{E}[(y-t)^2] &= \mathbb{E}[(y - y_\star)^2] + \mathrm{Var}(t) \\
&= \mathbb{E}[y_\star^2 - 2y_\star y + y^2] + \mathrm{Var}(t) \\
&= y_\star^2 - 2y_\star\mathbb{E}[y] + \mathbb{E}[y^2] + \mathrm{Var}(t) && \text{by linearity of expectation} \\
&= y_\star^2 - 2y_\star\mathbb{E}[y] + \mathbb{E}[y]^2 + \mathrm{Var}(y) + \mathrm{Var}(t) && \text{by the formula for variance} \\
&= \underbrace{(y_\star - \mathbb{E}[y])^2}_{\text{bias}} + \underbrace{\mathrm{Var}(y)}_{\text{variance}} + \underbrace{\mathrm{Var}(t)}_{\text{Bayes error}}
\end{aligned}$$

The first term is the **bias**, which tells us how far off the model's average prediction is. The second term is the **variance**, which tells us about the variability in its predictions as a result of the choice of training set, i.e. the amount to which it overfits the idiosyncrasies of the training data. The third term is the Bayes error, which we have no control over. So this decomposition is known as the **bias-variance decomposition**.

To visualize this, suppose we have two test examples, with targets $(t^{(1)}, t^{(2)})$. Figure 2 is a visualization in **output space**, where the axes correspond to the outputs of the model on these two examples. It shows the test error as a function of the predictions on these two test examples; because we're measuring mean squared error, the test error takes the shape of a quadratic bowl. The various quantities computed above can be seen in the diagram:

Understand why output space is different from input space or weight space.

- The generalization error is the average squared length $\|\mathbf{y} - \mathbf{t}\|^2$ of the line segment labeled *residual*.
- The bias term is the average squared length $\|\mathbb{E}[\mathbf{y}] - \mathbf{y}_*\|^2$ of the line segment labeled *bias*.
- The variance term is the spread in the green x's.
- The Bayes error is the spread in the black x's.

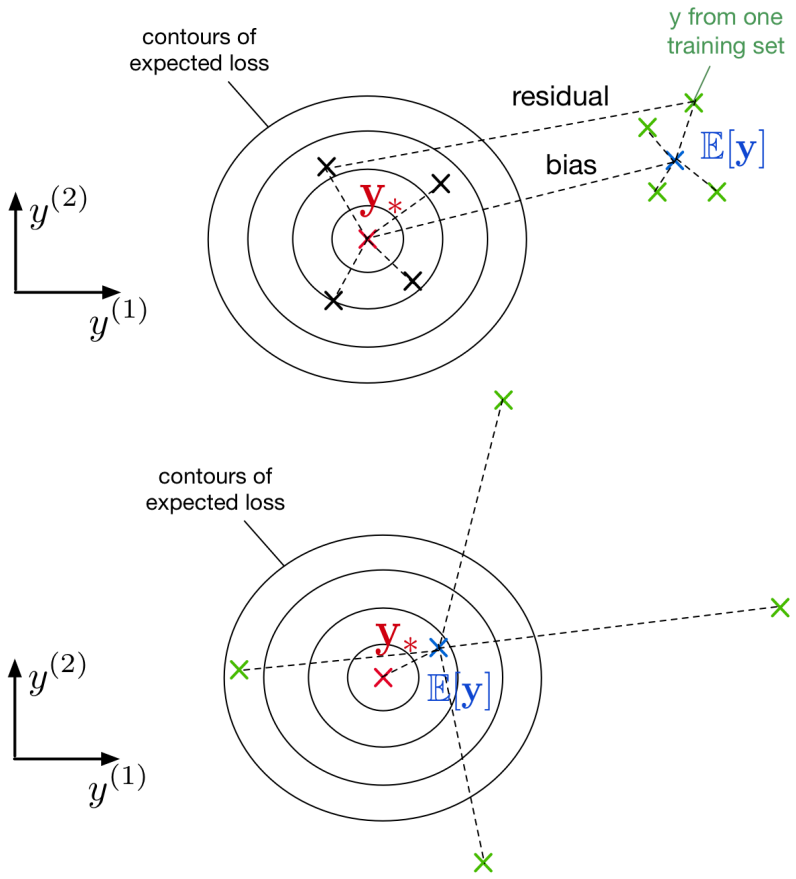

Figure 2: Schematic relating bias, variance, and error. **Top:** If the model is underfitting, the bias will be large, but the variance (spread of the green x's) will be small. **Bottom:** If the model is overfitting, the bias will be small, but the variance will be large.

4 Reducing overfitting

Now that we've talked about generalization error, let's review some strategies for improving generalization by reducing overfitting.

4.1 Reducing capacity

Remember the nonmonotonic relationship between model capacity and generalization error from Figure 1? We saw this effect in Lecture 2 when we increased degree M of the polynomial feature map in polynomial regression. This immediately suggests a strategy: tune the model capacity (e.g., the degree M in polynomial regression, or the number of layers / layer sizes in a neural network) on a validation set in order to find the sweet spot, which has enough capacity to learn the true regularities, but not enough to overfit.

Reducing capacity has an important drawback: it might make the model too simple to learn the true regularities in the data. Therefore, it's often preferable to keep the capacity high, but prevent it from overfitting in other ways. We'll discuss some such alternatives now.

4.2 Regularization and weight decay

So far, all of the cost functions we've discussed have consisted of the average of some loss function over the training set. Often, we want to add another term, called a **regularization term**, or **regularizer**, which penalizes hypotheses we think are somehow pathological and unlikely to generalize well. The total cost, then, is

$$\mathcal{J}(\boldsymbol{\theta}) = \underbrace{\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x},\boldsymbol{\theta}),t)}_{\text{training loss}} + \underbrace{\mathcal{R}(\boldsymbol{\theta})}_{\text{regularizer}} \tag{2}$$

For instance, suppose we are training a linear regression model with two inputs, x_1 and x_2, and these inputs are identical in the training set. The two sets of weights shown in Figure 3 will make identical predictions on the training set, so they are equivalent from the standpoint of minimizing the loss. However, Hypothesis A is somehow better, because we would expect it to be more stable if the data distribution changes. E.g., suppose we observe the input $(x_1 = 1, x_2 = 0)$ on the test set; in this case, Hypothesis A will predict 1, while Hypothesis B will predict -8. The former is probably more sensible. We would like a regularizer to favor Hypothesis A by assigning it a smaller penalty.

One such regularizer which achieves this is L_2 **regularization**; for a linear model, it is defined as follows:

This is an abuse of terminology; mathematically speaking, this really corresponds to the *squared* L_2 norm.

$$\mathcal{R}_{L_2}(\mathbf{w}) = \frac{\lambda}{2}\sum_{j=1}^{D} w_j^2. \tag{3}$$

L_2 regularization tends to favor hypotheses where the norms of the weights are smaller. For instance, in the above example, with $\lambda = 1$, it assigns a penalty of $\frac{1}{2}(1^2 + 1^2) = 1$ to Hypothesis A and $\frac{1}{2}((-8)^2 + 10^2) = 82$ to

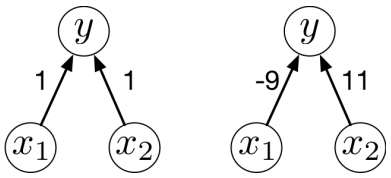

Figure 3: Two sets of weights which make the same predictions assuming inputs x_1 and x_2 are identical.

Hypothesis B, so it strongly prefers Hypothesis A. Because the cost function includes both the training loss and the regularizer, the training algorithm is encouraged to find a compromise between the fit to the training data and the norms of the weights. L_2 regularization can be generalized to neural nets in the obvious way: penalize the sum of squares of all the weights in all layers of the network.

It's pretty straightforward to incorporate regularizers into the stochastic gradient descent computations. In particular, by linearity of derivatives,

$$\frac{\partial \mathcal{J}}{\partial \theta_j} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \frac{\partial \mathcal{R}}{\partial \theta_j}. \tag{4}$$

If we derive the SGD update in the case of L_2 regularization, we get an interesting interpretation.

Observe that in SGD, the regularizer derivatives do not need to be estimated stochastically.

$$\begin{aligned}
\theta_j &\leftarrow \theta_j - \alpha \frac{\partial \mathcal{J}^{(i)}}{\partial \theta_j} && (5)\\
&= \theta_j - \alpha \left(\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \frac{\partial \mathcal{R}}{\partial \theta_j} \right) && (6)\\
&= \theta_j - \alpha \left(\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \lambda \theta_j \right) && (7)\\
&= (1 - \alpha\lambda)\theta_j - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}. && (8)
\end{aligned}$$

In each iteration, we shrink the weights by a factor of $1 - \alpha\lambda$. For this reason, L_2 regularization is also known as **weight decay**.

Regularization is one of the most fundamental strategies in machine learning. Regularizers are sometimes viewed as penalizing the "complexity" of a model, or favoring explanations which are "more likely."

4.3 Early stopping

Think about how the training and test error change over the course of training. Clearly, the training error ought to continue improving, since we're optimizing the training error. (If you find the training error going up, there may be something wrong with your optimizer.) The test error generally improves at first, but it may eventually start to increase as the model starts to overfit. Such a pattern is shown in Figure 4. (Curves such as these are referred to as **training curves**.) This suggests an obvious strategy: stop

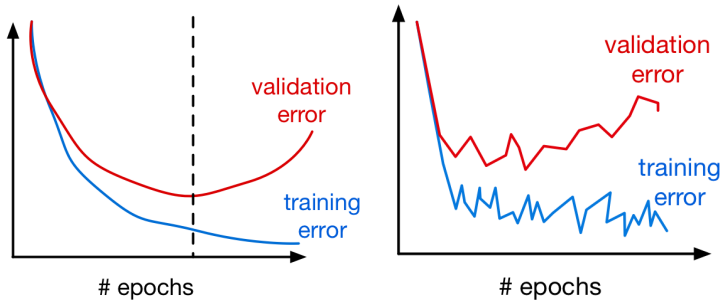

Figure 4: Training curves, showing the relationship between the number of training iterations and the training and test error. **(left)** Idealized version. **(right)** Accounting for fluctuations in the error, caused by stochasticity in the SGD updates.

the training at the point where the generalization error starts to increase. This strategy is known as **early stopping**. Of course, we can't do early stopping using the test set, because that would be cheating. Instead, we would determine when to stop by monitoring the validation error during training.

Unfortunately, implementing early stopping is a bit harder than it looks from this cartoon picture. The reason is that the training and validation error fluctuate during training (because of stochasticity in the gradients), so it can be hard to tell whether an increase is simply due to these fluctuations. One common heuristic is to space the validation error measurements far apart, e.g. once per epoch. If the validation error fails to improve after one epoch (or perhaps after several consecutive epochs), then we stop training. This heuristic isn't perfect, and if we're not careful, we might stop training too early.

4.4 Ensembles

Think back to Figure 2. If you average the predictions of multiple models trained independently on separate training sets, this reduces the variance of the predictions, which can lead to lower loss. Of course, we can't actually carry out the hypothetical procedure of sampling training sets independently (otherwise we're probably better off combining them into one big training set). We can try to simulate the effect of independent training sets by somehow injecting variability into the training procedure. Here some ways of injecting variability:

- Train on random subsets of the full training data. This procedure is known as **bagging**.
- Train different models or use different learning algorithms.

The set of trained models whose predictions we're combining is known as an **ensemble**. Ensembles often generalize quite a bit better than single models. This benefit is significant enough that the winning entries for most

of the major machine learning competitions (e.g. ImageNet, Netflix, etc.) used ensembles.

It's possible to prove that ensembles outperform individual models in the case of convex loss functions. In particular, suppose the loss function $\mathcal{L}$ is convex as a function of the outputs $\mathbf{y}$. Then, by the definition of convexity,

This isn't the same as the cost being convex as a function of $\boldsymbol{\theta}$. Lots of loss functions are convex with respect to $\mathbf{y}$, such as squared error or cross-entropy.

$$\mathcal{L}(\lambda_1 y_1 + \cdots + \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \cdots + \lambda_N \mathcal{L}(y_N, t) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1. \tag{9}$$

Hence, the average of the predictions must beat the average losses of the individual predictions. Note that this is true regardless of where the $\mathbf{y}$s came from. They could be outputs of different neural networks, or completely different learning algorithms, or even numbers you pulled out of a hat. The guarantee doesn't hold for non-convex cost functions (such as error rate), but ensembles still tend to be very effective in practice.

This result is closely related to the Rao-Blackwell theorem from statistics.