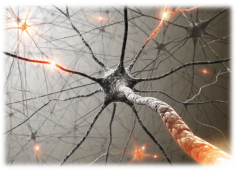
Machine learning: generalization



- In this module, I will talk about the generalization of machine learning algorithms.

Minimizing training loss

Hypothesis class:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

Training objective (loss function):

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

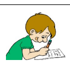
Optimization algorithm:

stochastic gradient descent

Is the training loss a good objective to optimize?

- Recall that our machine learning framework consists of specifying the hypothesis class, loss function, and the optimization algorithm.
- The hypothesis class could be linear predictors or neural networks. The loss function could be the hinge loss or the squared loss, which is averaged to produce the training loss.
- The default optimization algorithm is (stochastic) gradient descent.
- But let's be a bit more critical about the training loss. Is the training loss the right thing to optimize?

A strawman algorithm



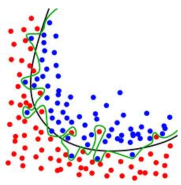
Algorithm: rote learning

Training: just store $\mathcal{D}_{\text{train}}$.
Predictor $f(x)$:
If $(x, y) \in \mathcal{D}_{\text{train}}$: return y.
Else: **segfault**.

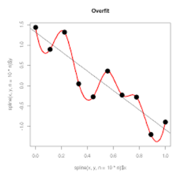
Minimizes the objective perfectly (zero), but clearly bad...

- Here is a strategy to consider: the rote learning algorithm, which just memorizes the training data and crashes otherwise.
- The rote learning algorithm does a perfect job of minimizing the training loss.
- But it's clearly a bad idea: It **overfits** to the training data and doesn't **generalize** to unseen examples.
- So clearly machine learning can't be about just minimizing the training loss.

Overfitting pictures



Classification



Regression

- This is an extreme example of **overfitting**, which is when a learning algorithm outputs a predictor that does well on the training data but not well on new examples.
- Here are two pictures that illustrate what overfitting looks like for binary classification and regression.
- On the left, we see that the green decision boundary gets zero training loss by separating all the blue points from the red ones. However, the smoother and simpler black curve is intuitively more likely to be the better classifier.
- On the right, we see that the predictor that goes through all the points will get zero training loss, but intuitively, the black line is perhaps a better option.
- In both cases, what is happening is that by over-optimizing on the training set, we risk fitting **noise** in the data.

Evaluation

$\mathcal{D}_{\text{train}} \longrightarrow$ learning algorithm $\longrightarrow f$

How good is the predictor f?

Key idea: the real learning objective

Our goal is to minimize **error on unseen future examples**.

Don't have unseen examples; next best thing:

Definition: test set

Test set $\mathcal{D}_{\text{test}}$ contains examples not used for training.

- So what is the true objective then? Taking a step back, machine learning is just a means to an end. What we're really doing is building a predictor to be deployed in the real world, and we just happen to be using machine learning. What we really care about is how accurate that predictor is on those **unseen future** inputs.
- Of course, we can't access unseen future examples, so the next best thing is to create a **test set**. As much as possible, we should treat the test set as a pristine thing that's unseen. We definitely should not tune our predictor based on the test set, because we wouldn't be able to do that on future examples.
- Of course at some point we have to run our algorithm on the test set, but just be aware that each time this is done, the test set becomes less good of an indicator of how well your predictor is actually doing.

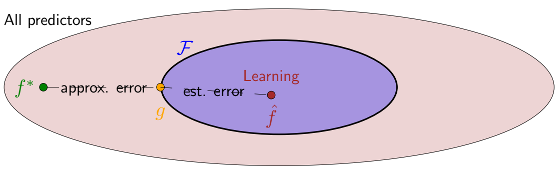
Generalization

When will a learning algorithm **generalize** well?

$\boxed{\mathcal{D}_{\text{train}}} \longrightarrow \boxed{\mathcal{D}_{\text{test}}}$

- So far, we have an intuitive feel for what overfitting is. How do we make this precise? In particular, when does a learning algorithm generalize from the training set to the test set?

Approximation and estimation error

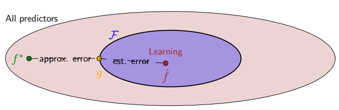


- Approximation error: how good is the hypothesis class?
- Estimation error: how good is the learned predictor **relative to** the potential of the hypothesis class?

$$\mathsf{Err}(\hat{f}) - \mathsf{Err}(f^*) = \underbrace{\mathsf{Err}(\hat{f}) - \mathsf{Err}(g)}_{\text{estimation}} + \underbrace{\mathsf{Err}(g) - \mathsf{Err}(f^*)}_{\text{approximation}}$$

- Here's a cartoon that can help you understand the balance between fitting and generalization. Out there somewhere, there is a magical predictor f^* that classifies everything perfectly. This predictor is unattainable; all we can hope to do is to use a combination of our domain knowledge and data to approximate that. The question is: how far are we away from f^*?
- Recall that our learning framework consists of (i) choosing a hypothesis class $\mathcal{F}$ (e.g., by defining the feature extractor) and then (ii) choosing a particular predictor $\hat{f}$ from $\mathcal{F}$.
- **Approximation error** is how far the entire hypothesis class is from the target predictor f^*. Larger hypothesis classes have lower approximation error. Let $g \in \mathcal{F}$ be the best predictor in the hypothesis class in the sense of minimizing test error $g = \arg\min_{f \in \mathcal{F}} \mathsf{Err}(f)$. Here, distance is just the differences in test error: $\mathsf{Err}(g) - \mathsf{Err}(f^*)$.
- **Estimation error** is how good the predictor $\hat{f}$ returned by the learning algorithm is with respect to the best in the hypothesis class: $\mathsf{Err}(\hat{f}) - \mathsf{Err}(g)$. Larger hypothesis classes have higher estimation error because it's harder to find a good predictor based on limited data.
- We'd like both approximation and estimation errors to be small, but there's a tradeoff here.

Effect of hypothesis class size



As the hypothesis class size increases...

Approximation error decreases because:

taking min over larger set

Estimation error increases because:

harder to estimate something more complex

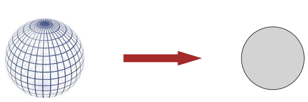
How do we control the hypothesis class size?

- The approximation error decreases monotonically as the hypothesis class size increases for a simple reason: you're taking a minimum over a larger set.
- The estimation error increases monotonically as the hypothesis class size increases for a deeper reason involving statistical learning theory (explained in CS229T).

Strategy 1: dimensionality

$$\mathbf{w} \in \mathbb{R}^d$$

Reduce the dimensionality d (number of features):



- Let's focus our attention to linear predictors. For each weight vector $\mathbf{w}$, we have a predictor $f_{\mathbf{w}}$ (for classification, $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$). So the hypothesis class $\mathcal{F} = \{f_{\mathbf{w}}\}$ is all the predictors as $\mathbf{w}$ ranges. By controlling the number of possible values of $\mathbf{w}$ that the learning algorithm is allowed to choose from, we control the size of the hypothesis class and thus guard against overfitting.
- One straightforward strategy is to change the dimensionality, which is the number of features. For example, linear functions are lower-dimensional than quadratic functions.

Controlling the dimensionality

Manual feature (template) selection:

- Add feature templates if they help
- Remove feature templates if they don't help

Automatic feature selection (beyond the scope of this class):

- Forward selection
- Boosting
- L_1 regularization

It's the number of features that matters

- Mathematically, you can think about removing a feature $\phi(x)_{37}$ as simply only allowing its corresponding weight to be zero ($w_{37} = 0$).
- Operationally, if you have a few feature templates, then it's probably easier to just manually include or exclude them — this will give you more intuition.
- If you have a lot of individual features, you can apply more automatic methods for selecting features, but these are beyond the scope of this class.
- An important point is that it's the number of features that matters, not the number of feature templates. (Can you define one feature template that results in severe overfitting?) Nor is it the amount of code that you have to write to generate a particular feature.

Strategy 2: norm

$$\mathbf{w} \in \mathbb{R}^d$$

Reduce the norm (length) $\|\mathbf{w}\|$:

- A related way to keep the weights small is called **regularization**, which involves adding an additional term to the objective function which penalizes the norm (length) of $\mathbf{w}$. This is probably the most common way to control the norm.

Controlling the norm

Regularized objective:

$$\min_{\mathbf{w}} \mathsf{TrainLoss}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$
For $t = 1, \ldots, T$:
$\quad\mathbf{w} \leftarrow \mathbf{w} - \eta(\nabla_{\mathbf{w}} \mathsf{TrainLoss}(\mathbf{w}) + \lambda \mathbf{w})$

Same as gradient descent, except shrink the weights towards zero by λ.

- This form of regularization is also known as L_2 regularization, or weight decay in deep learning literature.
- We can use gradient descent on this regularized objective, and this simply leads to an algorithm which subtracts a scaled down version of $\mathbf{w}$ in each iteration. This has the effect of keeping $\mathbf{w}$ closer to the origin than it otherwise would be.
- Note: Support Vector Machines are exactly hinge loss + L_2 regularization.

Controlling the norm: early stopping

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \dots, 0]$
For $t = 1, \dots, T$:
  $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

Idea: simply make T smaller

Intuition: if have fewer updates, then $\|\mathbf{w}\|$ can't get too big.

Lesson: try to minimize the training error, but don't try too hard.

- A really cheap way to keep the weights small is to do **early stopping**. As we run more iterations of gradient descent, the objective function improves. If we cared about the objective function, this would always be a good thing. However, our true objective is not the training loss.
- Each time we update the weights, $\mathbf{w}$ has the potential of getting larger, so by running gradient descent a fewer number of iterations, we are implicitly ensuring that $\mathbf{w}$ stays small.
- Though early stopping seems hacky, there is actually some theory behind it. And one paradoxical note is that we can sometimes get better solutions by performing less computation.

Summary

Not the real objective: training loss

Real objective: loss on unseen future examples

Semi-real objective: test loss

Key idea: keep it simple

Try to minimize training error, but keep the hypothesis class small.

- In summary, we started by noting that the training loss is not the objective. Instead it is minimizing unseen future examples, which is approximated by the test set provided you are careful.
- We've seen several ways to control the size of the hypothesis class (and thus reducing the estimation error) based on either reducing the dimensionality or reducing the norm.
- It is important to note that what matters is the **size** of the hypothesis class, not how "complex" the predictors in the hypothesis class look. To put it another way, using complex features backed by 1000 lines of code doesn't hurt you if there are only 5 of them.
- So far, we've talked about the various knobs that we can turn to control the size of the hypothesis class, but how much do we turn each knob?