salutary-overparameterization

January 9, 2022

1 SALUTARY OVERPARAMETERIZATION IN A SIMPLE LINEAR-REGRESSION SETTING

This is inspired by Peter Bartlett's benign-overfitting lecture, but it's not a direct demonstration of his claim. His claim is already proven, so I want to instead illustrate the essential idea in an extremely (read: "uselessly") simple setting, just to gain intuition.

1.1 Recap: What do we know about linear regression?

We know that the usual "ridgeless" OLS estimator gives the unique solution $[\,(RHS)'(RHS)\,]^{-1}\ (RHS)'y$ when R (the number of regressors, i.e. the number of the columns in the RHS matrix) is no larger than T (the number of observations), and gives a multiplicity of equally-well-fitting solutions when $R > T$. However, we also know that, despite training error dropping to zero as soon as $R \geq T$, the resulting "overfit" model does not generally hold up well out-of-sample. So, we can either reduce the dimensionality of the feature space (e.g. PCA on RHS, keeping only $R' < T$ of the resulting PC's of the regressors), or regularize the estimator with e.g. ridge.

1.1.1 Making the solution unique in the overparameterized case

Now, let's just tweak the "ridgeless" solution, and define it to be $\text{pinv}[\,(RHS)'(RHS)\,]\ (RHS)'y$ where pinv is the Moore-Penrose pseudoinverse. This gives a unique solution even when $R > T$, in particular it chooses, among all β's that fit the training data perfectly, the unique such β with smallest norm.

And, we'll also just observe that ridge regression (with positive regularization, that is, positive penalty factor) also has a well-specified and unique solution. Why? Well, the ridge of λ's added along the diagonal of $(RHS)'(RHS)$ turns "equivalent" (linearly-dependent) rows into linearly-independent rows, thereby making the matrix invertible.

1.2 Tautology: Implicit regularization is regularization

We're not going to dispute that when R is close to T, the ridgeless (indeed, even the ridge) solution holds up quite poorly out-of-sample. In fact, despite the fact that we've made the solution unique when $R > T$, we're not going to claim that either solution necessarily holds up well out-of-sample even in that case. However, we'll observe that as R becomes much larger than T, the ridgeless solution becomes indistinguishable from the ridge solution. This is not a shock, as we already characterized the Moore-Penrose pseudoinverse as giving rise to a "implicitly" regularized β.

1.3 Upshot: Benefit of aggressive overparameterization

But here's the shock: As R becomes much larger than T, the model begins to recover, and the $R \gg T$ cases give much, much smaller test error than the $R = T$ case! Here's what you should take away from this: The fact that highly "overparameterized" deep neural nets can achieve zero training error yet still perform well out-of-sample, might not be some mystery confined to deep learning only. We don't know for sure that the same pathways are active here, but we do see that even in the linear-regression setting, either an implicitly- or an explicitly-regularized solution is catastrophically bad out-of-sample when R (the number of parameters) is near T, yet recovers when $R \gg T$. So, maybe the key in deep learning is that it aggressively overparameterizes the model, and therefore gets good performance out-of-sample despite overfitting in-sample.

So, I go a step further than Bartlett: I don't call this "benign overfitting", but rather "salutary overparameterization"! Yes, the in-sample overfitting when the problem is overparameterized is benign: But I'd emphasize that the real shock is that the aggressive overparameterization itself recovers good out-of-sample accuracy!

[1]:
```
from typing import Tuple, Optional
import scipy.stats as stats
import pandas as pd
import numpy as np
from numpy.linalg import pinv
import sklearn.linear_model as lm
```

2 Data-generating process

Exceedingly simple. There are `N` raw-data feeds. Each feed is an i.i.d. standard Normal random variable. Of these, `N` feeds, `M` are actually relevant to our `y`-generating process, in the simplest way: `y` is the sum of the `M` relevant data feeds, plus white noise. Exactly which `M` are relevant is random. We have `T` observations in the training sample, and, because it doesn't matter, also `T` in the test sample.

We have $M \ll N$ and $T \lll N$. Now usually, we'd say that blindly using all `N` raw-data feeds is a terrible idea, because we'll be able to perfect interpolate (i.e. perfect overfit!) the training data. BUT… is that conventional wisdom actually true?

[2]:
```
DataSubsample = Tuple[pd.DataFrame, pd.Series]
# train, test
DataSample = Tuple[DataSubsample, DataSubsample]

N = 100_000
M = int(N * 0.01)
T = int(N * 0.001)
NOISE = 1


def _get_x_name(base: str="x", n: int=0) -> str:
    return f"{base}_{n}"
```

```
def ___gen_x(T: int=T) -> pd.Series:
    """Generate observations for a single raw-data feed."""
    x = stats.norm.rvs(size=T)
    x = pd.Series(x)
    return x

def __gen_X(N: int=N, T: int=T) -> pd.DataFrame:
    """Generate observations for all raw-data feeds."""
    X = {_get_x_name(n): ___gen_x() for n in range(N)}
    X = pd.DataFrame(X)
    assert X.shape == (T, N), X.shape
    return X

def _pick_relevant_x(N: int=N, M: int=M) -> pd.Series:
    """beta[n] indicates whether X[:,n] is relevant."""
    beta_star = pd.Series(0, index=range(N))
    # detereministically pick the first `M` of these, just so we
    # have the correct total number of "live" picks to distribute
    beta_star.iloc[:M] = 1
    # distribute the picks uniformly randomly
    beta_star = beta_star.sample(n=N).reset_index(drop=True)
    beta_star = beta_star.rename(index=_get_x_name)
    assert beta_star.sum() == M, beta_star.sum()
    return beta_star

def __get_y(beta_star: pd.Series, X: pd.DataFrame) -> pd.Series:
    """Generate y's, where beta_star indicates whether a column is relevant."""
    # pick out the relevant x's, zero out the rest
    y = X * beta_star
    # y is a simple sum of the relevant raw-data feeds
    y = y.sum(axis="columns")
    # add noise
    y = y + stats.norm.rvs(scale=NOISE, size=T)
    y.name = "y"
    return y


def _gen_data_subsample(beta_star: pd.Series) -> DataSubsample:
    """Get X, y."""
    X = __gen_X()
    y = __get_y(beta_star=beta_star, X=X)
    return X, y

def gen_data_sample() -> Tuple[pd.Series, DataSample]:
    beta_star = _pick_relevant_x()
    # train subsample
    X, y = _gen_data_subsample(beta_star=beta_star)
```

```
    train_subsample = X, y
    # test subsample
    X_, y_ = _gen_data_subsample(beta_star=beta_star)
    test_subsample = X_, y_
    sample = train_subsample, test_subsample
    return beta_star, sample

np.random.seed(1337)
# M relevant x's, training subsample, testing subsample
beta_star, ((X, y), (X_, y_)) = gen_data_sample()
```

3 Feature creation

To show how performance varies with increasing degree of overfitting, we need to increase the degree of overfitting. To do this in the linear regression setting, we can just increase the number of regressors until it equals—and indeed even further until it is much larger than—the number of observations. We'll do this in a very simple way: If we need `R` regressors, we'll use the first `R` columns of `X`. Notice that this doesn't matter: The `N` raw-data feeds were i.i.d. random, and which `M` of the `N` raw-data feeds were chosen to be relevant was also uniformly random. So shuffling the `N` raw-data feeds before choosing `R` of them is unnecessary.

[3]:
```
"""
# this implementation uses
# (a flavor of) random Fourier features,
# which would be much more useful for actual prediction.
# however, it's already getting to complicated
# for my liking,
# and IMO obscures the magic.

def _get_rhs(X: pd.DataFrame) -> pd.Series:
    w = stats.norm.rvs(size=len(X.columns))
    w = pd.Series(w, index=X.columns)
    rhs = X * w
    rhs = rhs.sum(axis="columns")
    rhs = np.sin(rhs)
    return rhs

def get_RHS(X: pd.DataFrame, R: int) -> pd.DataFrame:
    RHS = {_get_x_name(base="r", n=r): _get_rhs(X=X) for r in range(R)}
    RHS = pd.DataFrame(RHS)
    return RHS
"""

def get_RHS(X: pd.DataFrame, R: int) -> pd.DataFrame:
    return X.iloc[:, :R]
```

4 Model estimation

[4]:
```
RIDGE_REGULARIZATION = 1


def __fit_star_(**kwargs) -> pd.Series:
    """Ignores args and returns beta_star."""
    return beta_star

def __fit_ridgeless(y: pd.Series, RHS: pd.DataFrame) -> pd.Series:
    """Like ridge regression, but with 0 regularization.
    (X'X)^{-1} X'y, where ^{-1} denotes Moore-Penrose pseudoinverse.
    This is OLS if len(X.columns) <= len(y).
    """
    X_T_X = RHS.T @ RHS
    X_T_X_inv = pinv(X_T_X)
    X_T_y = RHS.T @ y
    beta_hat = X_T_X_inv @ X_T_y
    beta_hat = pd.Series(beta_hat, index=RHS.columns)
    return beta_hat

def __fit_ridge(
        y: pd.Series, RHS: pd.DataFrame,
        regularization: float=RIDGE_REGULARIZATION
    ) -> pd.Series:
    """This is OLS if penalty == 0."""
    model = lm.Ridge(alpha=regularization, fit_intercept=False)
    model = model.fit(y=y, X=RHS)
    beta_hat = pd.Series(model.coef_, index=RHS.columns)
    return beta_hat


def _fit_model(y: pd.Series, RHS: pd.DataFrame, kind: str="star") -> pd.Series:
    """Get beta_hat."""
    _fit = {
        "star": __fit_star_,
        "ridgeless": __fit_ridgeless,
        "ridge": __fit_ridge
    }[kind]
    beta_hat = _fit(y=y, RHS=RHS)
    return beta_hat

def fit_model(
        y: pd.Series, X: pd.DataFrame, R: int, kind: str="star"
    ) -> pd.Series:
    RHS = get_RHS(X=X, R=R)
    beta_hat = _fit_model(y=y, RHS=RHS, kind=kind)
```

```
    # fill the potentially-empty var's
    beta_hat = beta_hat.reindex(index=X.columns).fillna(0)
    return beta_hat

def fit_model_(R: int, kind: str="star"):
    return fit_model(y=y, X=X, R=R, kind=kind)
```

5 Model evaluation

[5]:
```
def __get_y_hat(beta: pd.Series, X: pd.DataFrame) -> pd.Series:
    y_hat = X * beta
    y_hat = y_hat.sum(axis="columns")
    return y_hat

def __get_loss(y: pd.Series, y_hat: pd.Series) -> float:
    """RMSE."""
    loss = y_hat - y
    loss = loss**2
    loss = loss.mean()
    loss = loss**0.5
    return loss

def _get_loss(y: pd.Series, X: pd.DataFrame, beta: pd.Series) -> float:
    y_hat = __get_y_hat(beta=beta, X=X)
    return __get_loss(y=y, y_hat=y_hat)

def get_loss(
        y: pd.Series, X: pd.DataFrame,  # train subsample
        y_: pd.Series, X_: pd.Series,  # test subsample
        R: int,
        kind: str="star"
    ) -> float:
    """Fit specified model and get test-subsample loss."""
    beta_hat = fit_model(y=y, X=X, R=R, kind=kind)
    loss = _get_loss(y=y_, X=X_, beta=beta_hat)
    return loss

def get_loss_(R: int, kind: str="star") -> float:
    return get_loss(y=y, X=X, y_=y_, X_=X_, R=R, kind=kind)
```

6 Results

In this oversimplified toy setup, the actual test-subsample performance of the models is almost indistinguishable from (indeed worse than) a model that simply ignores the x's and guesses 0 every time (`flat loss`), and nowhere near the performance of the ground-truth model (`star loss`).

Nevertheless, it shows the point: As $R \to T$, that is, as the number of regressors gets very close to

the number of observations, we suffer from a singularity issue, in particular when R approaches T from above. The RMSE of the OLS-type estimator blows up (`ridgeless loss`), and we see the canonical benefit of regularization (in this case, a no-thought ridge model) (`ridge loss`). Yet, when $R \gg T$, that is, when the number of regressors is much greater than the number of observations, the OLS-type model actually becomes indistinguishable from the regularized model. So, thanks to the Moore-Penrose pseudoinverse, as we really crank up the complexity (dimensionality) of our model, we get a natural implicit regularization, without ever having to consciously choose any regularization parameters.

But again, here's the real shocking takeaway: Conventional wisdom might suggest that, despite being unique, the solutions to the overparameterized fits ($R \gg T$) still won't hold up any better out-of-sample than the perfectly-parameterized fit ($R = T$). In both cases, we've perfectly overfit the training data. But the conventional wisdom is wrong... we know for a fact that the ridgeless model will give zero training error as soon as $R \geq T$ (and, if we make R big enough, so will ridge, as it finally gets "lucky" with a small-norm β that fits the training data perfectly). Yet, the $R \gg T$ cases give much, much smaller test error than the $R = T$ case!

Note: To actually get some good performance out of this, I think we could make the methodology closer to Bartlett's, e.g. use random Fourier features to have a better chance of "blending in" some of the predictive features.

[6]:
```
print("R=0:")
print(f"star loss = {get_loss_(R=None)}")
print(f"flat loss = {(y_**2).mean()**0.5}")
print()

for R in (1, 5, int(T/2), T-1, T, T+1, int((T+M) /2), M, M*2, M*5):
    print(f"R={R}:")
    for kind in ("ridgeless", "ridge"):
        print(f"{kind} loss = {get_loss_(R=R, kind=kind)}")
    print()
```

```
R=0:
star loss = 0.9236567098968085
flat loss = 34.854137423977086

R=1:
ridgeless loss = 34.69244635658993
ridge loss = 34.6932738017274

R=5:
ridgeless loss = 36.662583109678685
ridge loss = 36.63295490774653

R=50:
ridgeless loss = 49.33762691655742
ridge loss = 48.37308734314992

R=99:
```

```
ridgeless loss = 343.2648322926529
ridge loss = 68.69292052254256

R=100:
ridgeless loss = 2032.2315208423208
ridge loss = 68.17625896099608

R=101:
ridgeless loss = 526.7897819844312
ridge loss = 68.77494097796075

R=550:
ridgeless loss = 39.28105850975485
ridge loss = 39.26210071222908

R=1000:
ridgeless loss = 37.04166829684086
ridge loss = 37.036128880866926

R=2000:
ridgeless loss = 34.99921536904601
ridge loss = 34.99878478966434

R=5000:
ridgeless loss = 34.58332111001222
ridge loss = 34.61142971092663
```