1 Module 15: Gradient Descent Optimization

Contents

1.1 1D Gradient Descent

Gradient descent works on any arbitrary function/surface, to find its minimum.

In 1D, for some learning rate α:

$$x^{(t+1)} = x^{(t)} - \alpha \frac{d}{dx} f(x)$$

1.1.1 Linear Regression example

For example, taking themean squared error $\frac{1}{N}\sum_i^N(\hat{y}_i - y_i)^2$ of some linear model $\hat{y} = \theta \cdot x$. Note, we are optimizing the parameter for θ, not x. Therefore, the function we optimize (derivative of MSE) is $\frac{d}{d\theta} = \frac{1}{N}\sum_i^N 2 \cdot (\hat{y}_i - y_i) \cdot x$.

The mathematical representation for optimizing θ is then

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{d}{d\theta} L(\theta^{(t)}, \mathbb{X}, \hat{y})$$

for a loss function L with the dataset $\mathbb{X}$ and the model $\hat{y}$.

In Scipy and Sklearn, fitting is performed using gradient descent because the amt. of data is much smaller.

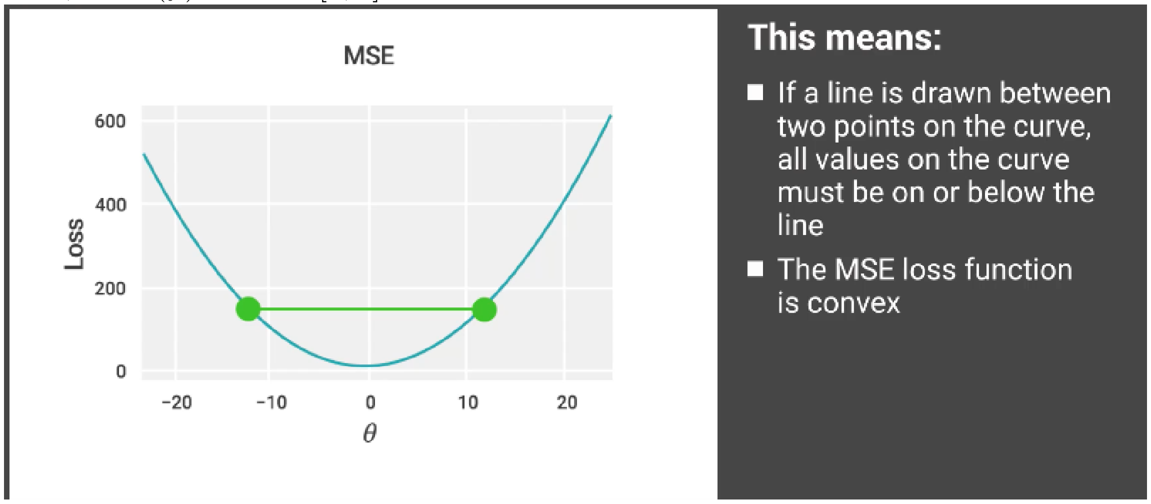
```
function = lambda x: (x^4 - 15x^3 + 80x^2 - 180x + 144) / 10
x = np.linspace(1, 6.75, 200)

from scipy.optimize import minimize
minimize(function, x0 = 6)
# x0 is the starting location. For the above function, for example,
minimize(function, x0 = 6) != minimize(function, x0 = 1)
```

1.1.2 Convexity

Defined by the inequality formula

$$t \cdot f(a) + (1-t) \cdot f(b) \leq f(t \cdot a + (1-t) \cdot b)$$

for all $a, b \in \mathbb{D}(f)$ and $t \in [0, 1]$



This means:

- If a line is drawn between two points on the curve, all values on the curve must be on or below the line
- The MSE loss function is convex

The 1-dimensional MSE function is always convex.

1.2 2D minimization for Linear Regression

In the case of fitting a linear regression, the second parameter is the intercept: $\hat{y} = \theta_0 + \theta_1 \cdot x$

With linear regressions, we can fit without intercept by introducing a bias column of 1s and fitting it. The bias column will represent the intercept which can be fit.

```
# X has X["data"]
X["bias"] = 1
# Therefore X = [[1,data_1], [1,data_2], etc.]

model = LinearRegression(fit_intercept=False)
model.fit(X,y)
model.coef_ # --> returns a tuple e.g. (0.9, 0.1) = £(\theta_0,\theta_1)£
# In which case we know that 0.9 is the intercept

# Predict with
\theta_0 * X.iloc[:,0] + \theta_1 * X.iloc[:,1]
# OR
X @ np.array([\theta_0, \theta_1])
```

1.3 Multidimensional Gradient Descent

In 2D, our derivative will be 2D; e.g.

$$\begin{aligned} &\text{arbitrary function} f(\theta_0, \theta_1) = 8\theta_0^2 + 3\theta_0\theta_1 \\ &\nabla_{\vec{\theta}} f = \frac{df}{d\theta_0}\hat{i} + \frac{df}{d\theta_1}\hat{j} = \begin{bmatrix} 16\theta_0 + 3\theta_1 \\ 3\theta_0 \end{bmatrix} \end{aligned} \tag{1}$$

As such, with p+1 variables, the gradient is

$$\nabla_{\vec{\theta}} f(\vec{\theta}) = \frac{df}{d\theta_0}\hat{i} + \frac{df}{d\theta_1}\hat{j} = \begin{bmatrix} \frac{d}{d\theta_0}(f) \\ \frac{d}{d\theta_1}(f) \\ \vdots \\ \frac{d}{d\theta_p}(f) \end{bmatrix}$$

To be used with the gradient descent algorithm

$$\vec{\theta}^{(t+1)} - \alpha \nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})$$

1.4 Stochastic Gradient Descent

If the dataset is large, then computing the gradient is computationally expensive. Instead, compute on successive subsets of the dataset which only technically *approximate* the gradient (i.e. SGD is not globally optimal; not the most efficient), but are far less expensive.

Gradient Descent

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \rho(\tau) \left(\frac{1}{n} \sum_{i=1_\tau}^{n} \nabla_\theta L_i(\theta) \Big|_{\theta=\theta(\tau)} \right)$$

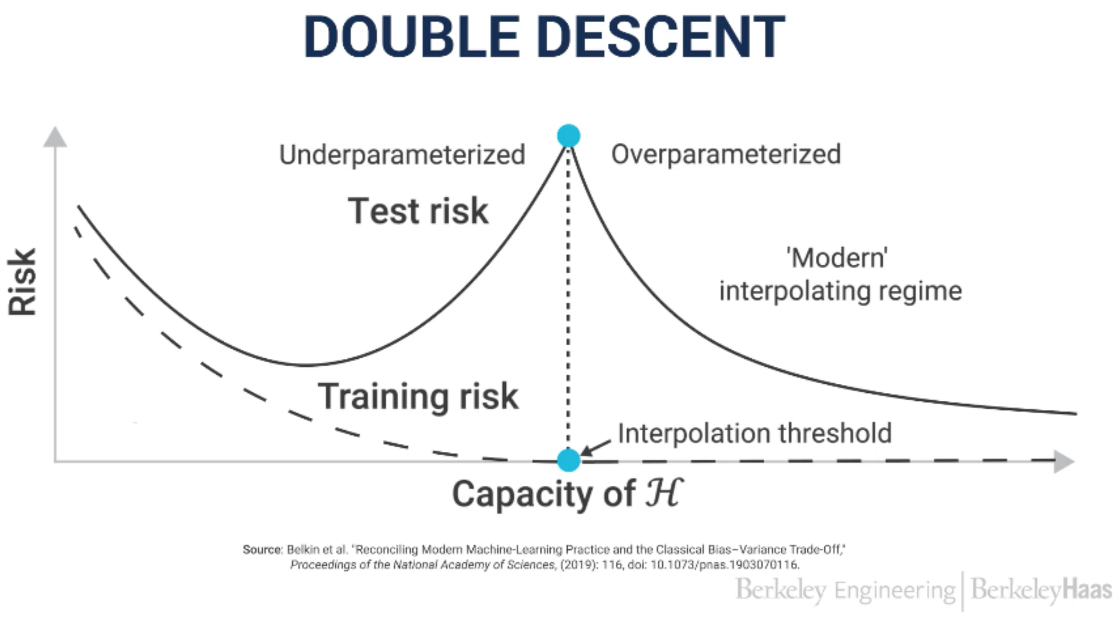
vs. Stochastic Gradient Descent

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \rho(\tau) \left(\frac{1}{\mathbb{B}} \sum_{i\in\mathbb{B}}^{\mathbb{B}} \nabla_\theta L_i(\theta) \Big|_{\theta=\theta(\tau)} \right)$$

Essentially, we don't compute the full loss surface per iteration; instead, we compute a subset and allow successive iterations to converge towards the solution.

1.5 Double Descent (Implicit Regularization)

Increasing model complexity well beyond the initial test error minimum may eventually slowly descend in test error again. Specifically, this occurs when training error is 0 (”Modern Interpolating Regime”).

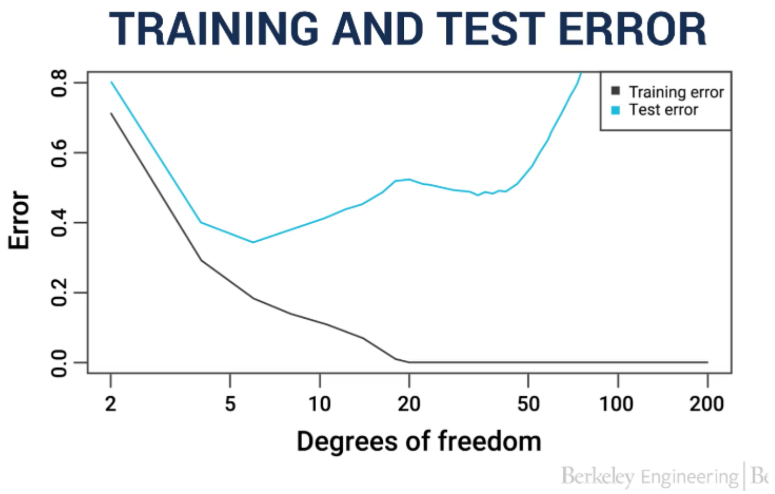


where ”Capacity of $\mathbb{H}$” == Model Complexity and ”Risk” == ”Error”

In lieu of formal mathematical definitions, allow

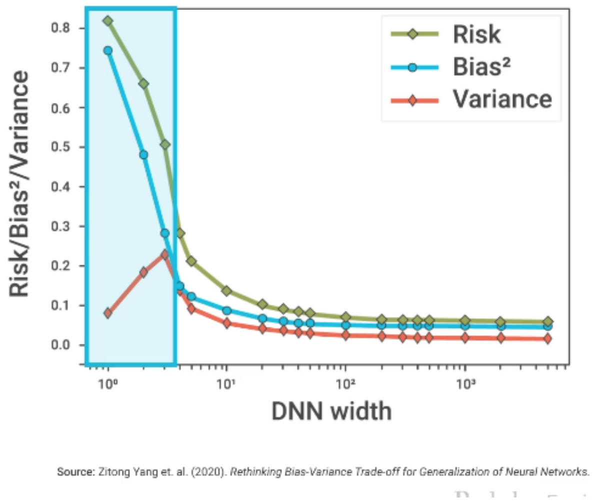
$$\text{model risk} = \sigma^2 + (\text{model bias}^2) + \text{model variance}$$

This rule works to some effect from large neural nets to even simplistic linear regressions (with high polynomial number). It is more pronounced in very large neural nets.



Alternatively, there may just be another minima (not necessarily better than the first), such as with this example which a Spline Linear Regression fitted with 20 data points (degrees of freedom → polynomial order of linear regression).

In this example with a DNN, the expected solution lies in the classical regime (highlighted blue). But the most optimal answer is in the regime where the complexity is much larger.



So modern practice uses gigantic neural networks with number of parameters >> available data.

WHY?: Active topic of research but – after the complexity > number of data points, there are an infinite number of solutions with 0 training error. As such, the fitting method – as in, Stochastic Gradient Descent – experimentally "chooses" the optimal model with 0 training error. This solution is *implicitly regularized* i.e. there is no explicit regularization penalty.

2021 Barrett Paper on the subject: `https://arxiv.org/abs/2009.11162`