Further into AI:
Dealing With OverFitting & Intro To Optimization

Minor In AI, IIT Ropar

9th April, 2025

Welcome back!

1 Taming the Beast: Regularization, Gradients, and the Path to Optimal Learning

A Tale of Two Gardeners (Introduction to Overfitting)

Imagine two gardeners, Sia and Deep. They both want to grow prize-winning tomatoes. Sia, being a meticulous gardener, analyzes every single tomato plant with intense scrutiny. She measures the soil pH, the amount of sunlight each leaf receives, and the precise angle of every stem. She adjusts her watering and fertilizing schedule based on these micro-measurements. Her tomatoes in the first season are AMAZING! They are big, juicy, and win every prize at the local fair.

Deep, on the other hand, takes a more relaxed approach. He waters regularly, makes sure the plants get enough sun, and uses a general-purpose fertilizer. He doesn't get caught up in the tiny details. His tomatoes are good, but not quite as spectacular as Sia's in the first season.

Figure 1: A cartoon comparing Sia's overly complex setup with Deep's simple, effective approach.

But then comes the second season. Sia, confident in her hyper-detailed approach, uses the *exact* same methods. To her surprise, her tomatoes are smaller, less flavorful, and prone to diseases. Deep's tomatoes, however, are just as good as the previous year.

What happened? Sia *overfit* to the specific conditions of her garden in the first season. She learned the noise and quirks of that particular year, and her methods didn't generalize well. Deep, by focusing on the broader principles of tomato growing, built a more *robust* and adaptable system.

This story illustrates a crucial concept in machine learning: *overfitting*. Just like Sia, our models can become too specialized to the training data and perform poorly on new, unseen data. This chapter is all about learning how to prevent this.

1.1 Revisiting Linear Regression and the Problem of Overfitting

We've already touched upon linear regression, a powerful tool for finding relationships in data. But as we build more complex models (like polynomial regression), we risk overfitting. Our goal is to find the sweet spot – a model that captures the underlying patterns without memorizing the noise.

1.2 Ridge and Lasso Regression: The Regularization Rangers

Enter Ridge and Lasso Regression, two powerful techniques to prevent overfitting. Think of them as adding constraints, or penalties, to the learning process. These "penalties" discourage the model from assigning excessively large coefficients to features. These large coefficients are often a sign of overfitting, where the model is too sensitive to small fluctuations in the training data.

1.2.1 Ridge Regression (L2 Regularization)

Ridge Regression adds a penalty proportional to the *square* of the magnitude of the coefficients. This forces the coefficients to be smaller overall, but rarely forces them to be exactly zero.

1.2.2 Lasso Regression (L1 Regularization)

Lasso Regression, on the other hand, adds a penalty proportional to the *absolute value* of the coefficients. A key difference is that Lasso can force some coefficients to be *exactly zero*, effectively eliminating those features from the model. This makes Lasso useful for feature selection.

Code in Action:

Let's see these techniques in action. We'll start with a simple dataset.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_squared_error, r2_score

# Sample dataset (6 points, 2 features)
X = np.array([
    [1, 1],
    [2, 2],
    [3, 3],
    [4, 4],
    [5, 5],
    [6, 6]
])

# Target variable (roughly y = 1*x1 + 1*x2)
y = np.array([3, 5, 7, 9, 11, 13])

```

```
# Train all three models
lr = LinearRegression().fit(X, y)
ridge = Ridge(alpha=1.0).fit(X, y)
lasso = Lasso(alpha=1.0).fit(X, y)

# Predictions
y_pred_lr = lr.predict(X)
y_pred_ridge = ridge.predict(X)
y_pred_lasso = lasso.predict(X)

# Print metrics
print("=== Linear Regression ===")
print("Coefficients:", lr.coef_)
print("Intercept:", lr.intercept_)
print("MSE:", mean_squared_error(y, y_pred_lr))
print("R^2 Score:", r2_score(y, y_pred_lr))

print("\n=== Ridge Regression ===")
print("Coefficients:", ridge.coef_)
print("Intercept:", ridge.intercept_)
print("MSE:", mean_squared_error(y, y_pred_ridge))
print("R^2 Score:", r2_score(y, y_pred_ridge))

print("\n=== Lasso Regression ===")
print("Coefficients:", lasso.coef_)
print("Intercept:", lasso.intercept_)
print("MSE:", mean_squared_error(y, y_pred_lasso))
print("R^2 Score:", r2_score(y, y_pred_lasso))
```

Listing 1: Ridge and Lasso Regression Comparison

In this code:

- We create a simple dataset with two features and a target variable.
- We train three models: Linear Regression, Ridge Regression (with `alpha=1.0`), and Lasso Regression (with `alpha=1.0`). The `alpha` parameter controls the strength of the regularization penalty. Larger `alpha` means stronger regularization.
- We print the coefficients, Mean Squared Error (MSE), and R-squared (R^2) score for each model. Notice how the coefficients differ between the models. Ridge and Lasso will generally have smaller coefficients than standard Linear Regression.

1.2.3 Evaluating Performance with R-squared and MSE

- **Mean Squared Error (MSE):** Measures the average squared difference between the predicted values and the actual values. Lower MSE indicates better performance.
- **R-squared (R^2):** Represents the proportion of variance in the dependent variable that can be predicted from the independent variables. It ranges from 0 to 1, with higher values indicating a better fit.

It's important to note that a lower MSE on the training data doesn't *always* mean a better model. Overfitting can lead to a very low training MSE but poor performance on new data. That's why we need to consider R-squared and, more importantly, evaluate performance on a separate *test set* (which we'll cover in later chapters).

1.2.4 Making Predictions

```python
# === Step 6: Predict on New Input ===
new_sample = np.array([[7, 7]])

pred_lr = lr.predict(new_sample)
pred_ridge = ridge.predict(new_sample)
pred_lasso = lasso.predict(new_sample)

print("\n=== Prediction for New Input [7, 7] ===")
print("Linear Regression Prediction:", pred_lr[0])
print("Ridge Regression Prediction:", pred_ridge[0])
print("Lasso Regression Prediction:", pred_lasso[0])
```

Listing 2: Making Predictions with Trained Models

This shows how to use the trained models to predict the target variable for new, unseen data.

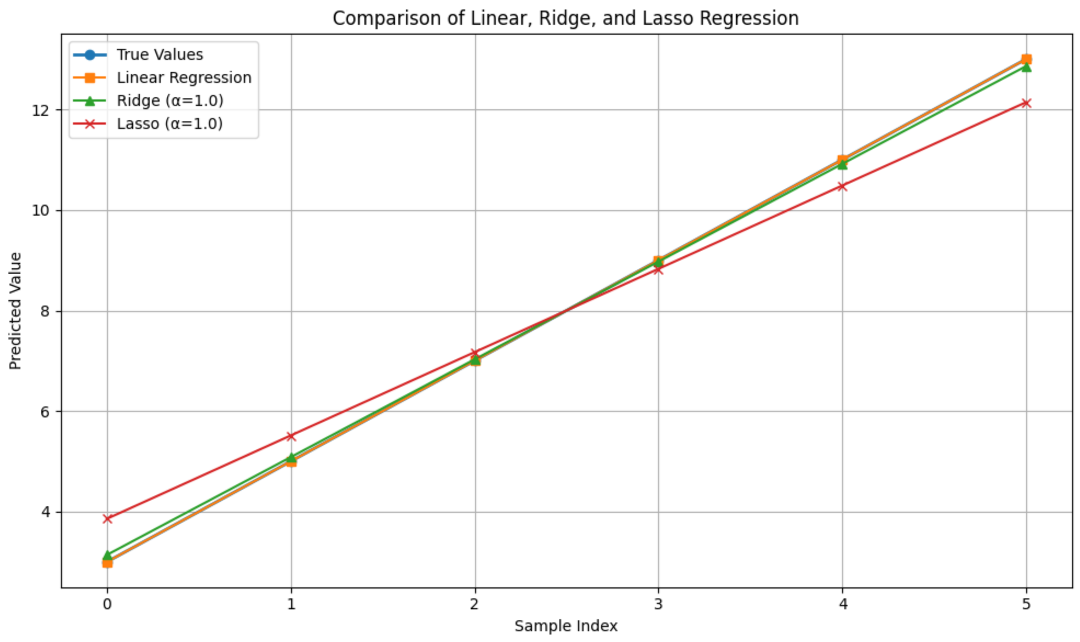

Figure 2: A plot comparing the predictions of Linear, Ridge, and Lasso regression on the same data, visually showing how Ridge and Lasso predictions are "shrunk" compared to Linear Regression.

1.3 Finding the Lowest Point: Introduction to Cost Functions and Gradients

Now, let's shift gears and talk about how these models *learn*. The core idea is to find the "best" set of coefficients that minimize a *cost function*. The cost function measures how well the model is performing. The goal is to find the parameters (coefficients) that minimize the cost.

1.3.1 The Cost Function: A Landscape Metaphor

Think of the cost function as a landscape. The height of the landscape represents the cost (error), and the location represents the model's parameters (coefficients). Our goal is to find the lowest point in this landscape.

For regression problems, Mean Squared Error (MSE) is a commonly used cost function. Simply put, MSE measures the average "squared distance" between what our model predicts and what the actual values are. The larger this distance, the worse our model is performing.

The MSE landscape for a model with two parameters might look like a bowl or valley. Our goal is to find the bottom of this valley - the point where the prediction error is minimized.

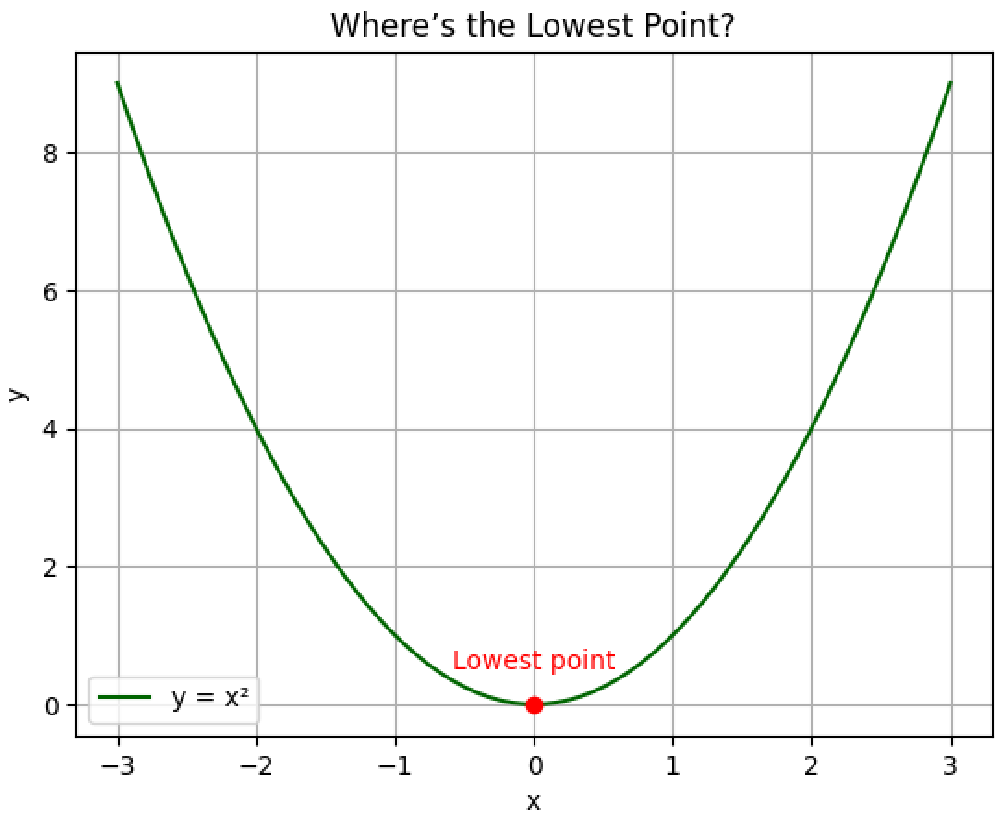

Figure 3: A bowl-shaped cost function. Our goal is to find the lowest point.

1.3.2 The Gradient: Nature's Compass

Imagine you're blindfolded in this landscape. How would you find your way to the lowest point? You would feel the ground beneath your feet to determine which direction slopes downward the most steeply, then take a step in that direction.

This is exactly what the *gradient* does. Think of the gradient as an arrow that:

1. Points in the direction of the steepest uphill climb
2. Has a length proportional to how steep that climb would be

Since we want to go downhill (minimize the cost), we move in the exact opposite direction of the gradient.

1.3.3 Gradient Descent: A Hiking Story

Imagine three hikers - Ana, Bo, and Cal - all trying to reach the bottom of a valley in thick fog:

Ana's Approach (Small Steps): Ana is cautious and takes tiny steps downhill. She carefully feels the slope beneath her feet and moves slowly in the direction that goes downward. She's unlikely to miss the bottom, but her journey takes a very long time.

Bo's Approach (Medium Steps): Bo takes reasonably sized steps. He still pays attention to the slope beneath his feet but moves with more confidence. He makes good progress and still manages to navigate the terrain safely.

Cal's Approach (Large Steps): Cal is impatient and takes huge leaps. Sometimes this works well when he's far from the bottom, but often he overshoots, ending up on the opposite slope and having to zigzag back and forth, or worse, leaping completely out of the valley.

This hiking analogy illustrates gradient descent, an iterative process where we:

1. Start at some position on the "error landscape"
2. Feel the slope (compute the gradient)
3. Take a step downhill (in the opposite direction of the gradient)
4. Repeat until we reach the bottom (or get close enough)

1.3.4 The Learning Rate: Finding the Right Step Size

The *learning rate* is essentially how big of a step we take each time. It's a crucial setting in gradient descent:

- **Too small (like Ana):** We take tiny steps and the journey to the minimum takes forever. Training might take an impractically long time.
- **Too large (like Cal):** We take huge steps and might overshoot the minimum, potentially bouncing back and forth between slopes or even walking away from the solution entirely.
- **Just right (like Bo):** We make steady progress toward the minimum, converging in a reasonable amount of time.

The ideal learning rate varies depending on the specific problem and dataset. Finding the right learning rate often involves experimentation - like trying different step sizes until you find one that works well.

Learning Rate Comparison

Learning Rate	**Pros**	**Cons**
Small (Ana)	Stable, won't overshoot	Slow convergence
Medium (Bo)	Good balance	May still be slow for some problems
Large (Cal)	Fast when it works	May oscillate or diverge

1.3.5 The Journey to the Bottom

Imagine we're training a simple linear regression model. Here's what the journey looks like:

1. We start with a guess for our model's parameters (like throwing a ball randomly into the valley).
2. We check how steep the ground is at that point (compute the gradient).
3. We take a step downhill (update our parameters).
4. We keep repeating steps 2 and 3 until we reach the bottom.

With each step, our model gets better at making predictions. The cost (error) decreases, and we get closer to the optimal solution.

1.3.6 Convergence: Are We There Yet?

How do we know when we've reached the bottom? Here are some ways to tell:

1. The ground becomes nearly flat (the gradient becomes very small).
2. We're not making meaningful progress anymore (the cost isn't decreasing much).
3. We've taken a predefined number of steps and decide that's enough.

The beauty of gradient descent is its simplicity and effectiveness. By repeatedly stepping downhill, we can find good solutions to complex problems, even when dealing with models that have many parameters to optimize.

This chapter has introduced you to some essential concepts in machine learning: regularization to prevent overfitting, cost functions to quantify model performance, and gradient descent to find optimal model parameters. By understanding these concepts, you're well on your way to building robust and effective machine learning models. The journey is just beginning, but you've taken some important first steps. Keep exploring, keep experimenting, and keep learning!