Linear Regression with Gradient Descent

Latreche Sara

June 9, 2025

Contents

1 Introduction

Linear regression models the relationship between an input variable x and an output variable y by fitting a straight line:

$$\hat{y} = wx + b$$

Our goal is to find parameters w (weight) and b (bias) that minimize the difference between predicted and actual values.

2 Linear Regression: Learning from Data

Linear regression is a fundamental statistical technique used to understand the relationship between two variables. In the context of **machine learning (ML)**, it helps us model the relationship between **features** (inputs) and a **label** (output).

Real-World Example: Predicting Weight from Height

Suppose you want to predict a person's **weight** (in kilograms) based on their **height** (in centimeters). You collect the following data:

Height (cm)	**Weight (kg)**
150	50
160	55
170	65
180	72
190	80

When you plot this data, you'll likely observe a **positive trend**— taller people generally weigh more.

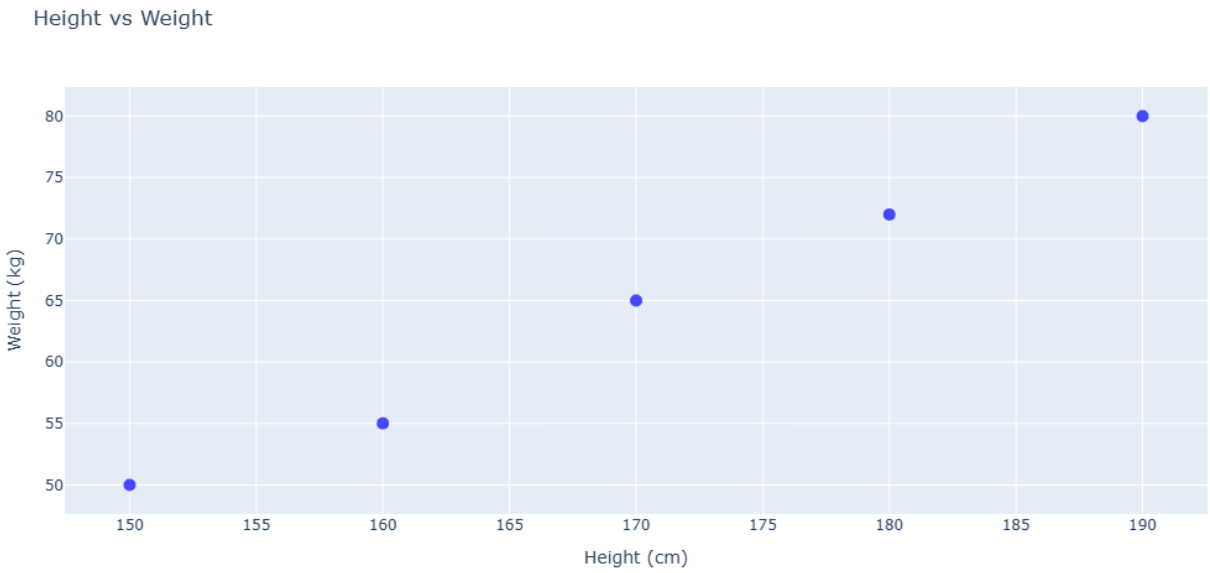


Figure 1: Scatter Plot of Height vs Weight (Without Best Fit Line)

We could create our own model by drawing a best fit line through the points:

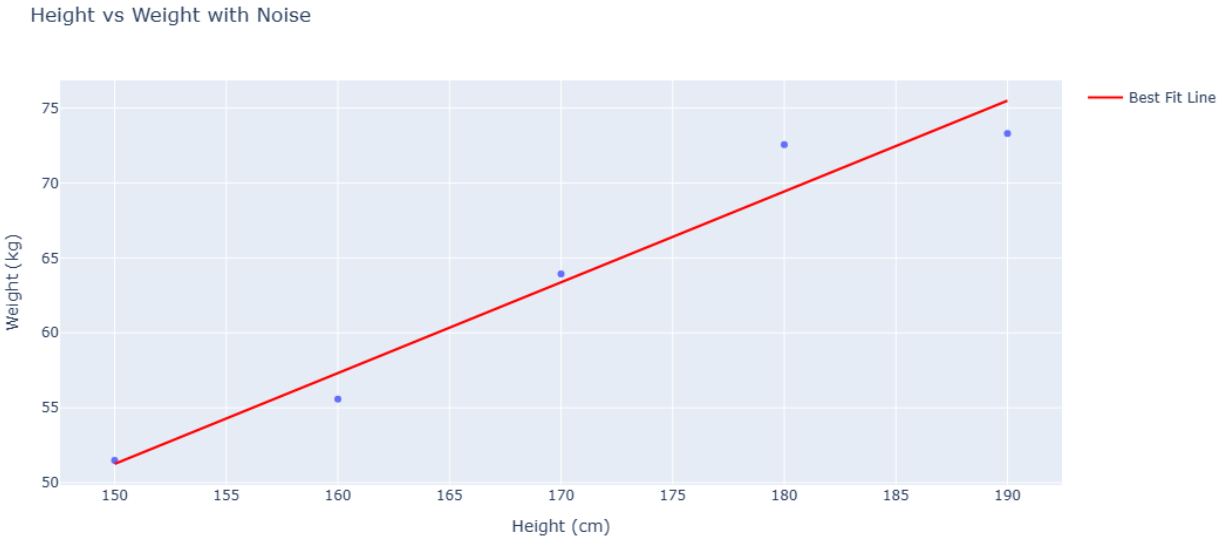


Figure 2: Height vs Weight with Best Fit Line

Linear regression helps us fit a straight line that best represents this relationship. The model takes the form of a simple equation:

$$\hat{y} = w \cdot x + b$$

Where:

- $\hat{y}$ is the predicted **weight**,
- x is the input **height**,
- w is the **slope** of the line (also called the weight of the model),
- b is the **bias** or **intercept**—where the line crosses the y-axis.

Example Model

Let's say we trained a linear model and found:

$$w = 0.6 \quad \text{and} \quad b = -40$$

Then the model becomes:

$$\hat{y} = 0.6 \cdot x - 40$$

So, for a person who is 170 cm tall:

$$\hat{y} = 0.6 \cdot 170 - 40 = 62 \text{ kg}$$

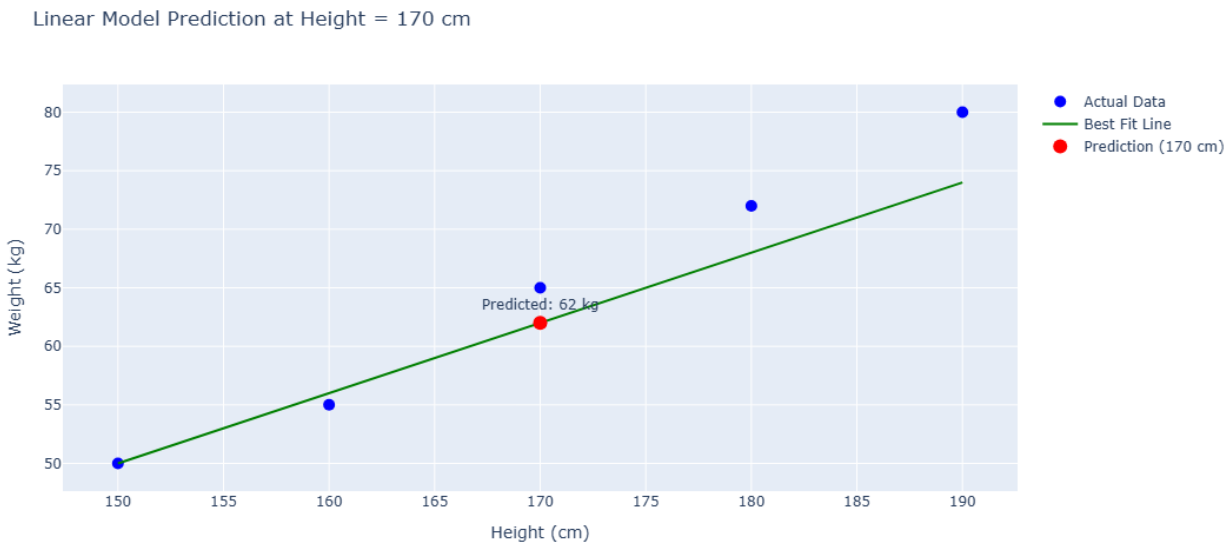


Figure 3: Prediction for Height = 170 cm Using the Learned Line

3 Linear Regression Equation in Machine Learning

In **algebraic terms**, the linear regression model can be defined as:

$$y = w \cdot x + b$$

Where:

- y is the predicted value, which in the example might represent the *miles per gallon* of a car.
- x is the input value, such as the *weight of the car* in pounds.
- w is the *slope* of the line, indicating how much y changes for each unit change in x.
- b is the *y-intercept*, representing the value of y when $x = 0$.

In the context of **Machine Learning (ML)**, we express the equation for linear regression using predicted values:

$$\hat{y} = w \cdot x + b$$

Where:

- $\hat{y}$ is the *predicted label* or output of the model.
- x is the *input feature* (e.g., car weight).
- w is the *weight* of the feature, learned during training. It reflects the strength and direction of the relationship between the feature and the output.
- b is the *bias* term (same role as the y-intercept). In ML, it shifts the prediction line up or down to better fit the data.

4 Models with Multiple Features

So far, we've used only one feature—like a person's height—to predict weight using linear regression.

But in real-world scenarios, predictions often depend on **multiple features**, such as age, gender, and lifestyle factors.

General Form of the Model

If a model uses multiple features, say $x_1, x_2, x_3, \ldots, x_n$, the linear regression equation becomes:

$$\hat{y} = b + w_1 x_1 + w_2 x_2 + w_3 x_3 + \cdots + w_n x_n$$

Where:

- $\hat{y}$ is the **predicted output** (e.g., weight in kg)
- b is the **bias** (intercept)
- $w_1, w_2, \ldots, w_n$ are the **weights** for each feature
- $x_1, x_2, \ldots, x_n$ are the **input features**

Example: Predicting Weight Based on Multiple Features

Suppose we want to predict a person's **weight** based on:

Height (cm)	**Age (years)**	**Shoe Size (EU)**	**Weight (kg)**
150	22	38	50
160	25	39	55
165	30	40	60
170	28	42	65
175	35	43	68
180	40	44	72
185	36	45	78
190	50	46	83

Let's define the features:

- x_1: Height (in cm)
- x_2: Age (in years)
- x_3: Shoe Size

Visualizing Feature Relationships

By plotting these additional features against weight, we observe they exhibit linear trends:

Figure 2: Age vs Weight

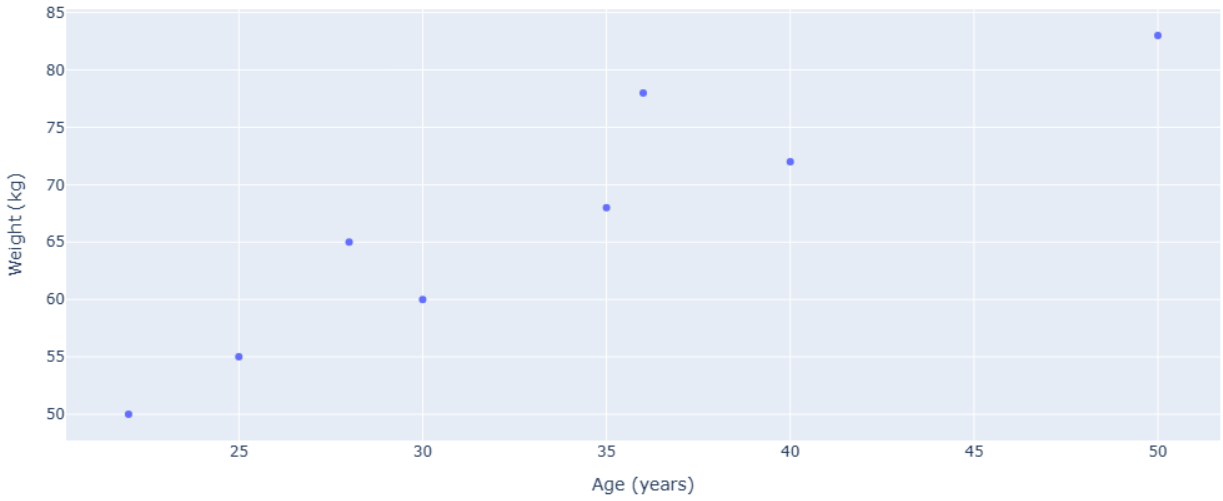
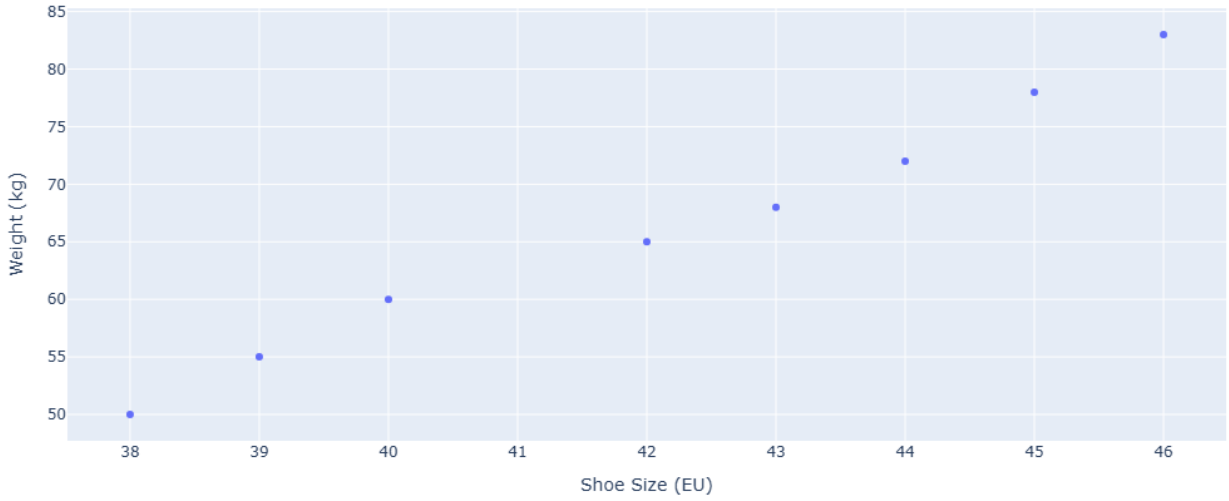


Figure 3: Shoe Size vs Weight



5 Linear Regression Model

After training, we found the following parameters:

- $w_1 = 0.393$ (Height coefficient)
- $w_2 = 0.143$ (Age coefficient)
- $w_3 = 0.882$(Shoe Size coefficient)
- $b = -58.492$ (Bias term)

So, our linear model is:

$$\hat{y} = 0.393 \cdot \text{Height} + 0.143 \cdot \text{Age} + 0.882 \cdot \text{ShoeSize} - 58.492$$

Example Prediction

Suppose a person has the following characteristics:

- Height: 170 cm
- Age: 28 years
- Shoe Size: 42 EU

Then the predicted weight is:

$$\hat{y} = 0.393 \cdot 170 + 0.143 \cdot 28 + 0.882 \cdot 42 - 58.492$$

$$\hat{y} = 66.81 + 4.004 + 37.044 - 58.492 = 49.366 \text{ kg}$$

In this setup, linear regression learns separate **weights** (w_i) for each feature to best predict the target output (e.g., weight). This is known as **multiple linear regression**, and it allows the model to capture more complex patterns than simple linear regression with one feature.

6 Linear Regression: Loss

In machine learning, **loss** is a key concept that helps in evaluating the performance of a model. Specifically for linear regression, the goal is to minimize the difference between the predicted values and the actual target values, i.e., the labels.

Loss in Linear Regression

In linear regression, we aim to fit a line to the data points so that the predictions of the model are as close as possible to the actual data points. The loss function quantifies the error (the difference) between the actual data points and the model's predictions.

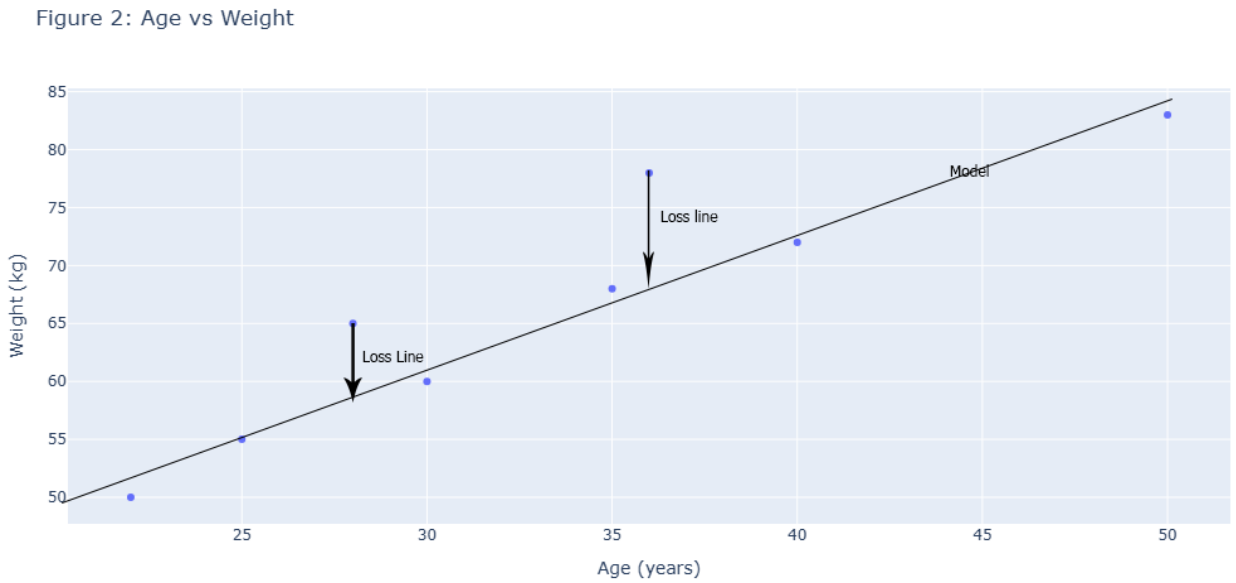


Figure 4. Loss is measured from the actual value to the predicted value.

In some textbooks, the loss function is also called the **cost function** and is often denoted by the symbol J. This function represents the overall error of the model and serves as the objective we want to minimize during training.

6.1 Understanding Loss as Distance

In machine learning and statistics, **loss** quantifies how far off a model's prediction is from the actual value. Rather than focusing on whether the model overestimated or underestimated, loss measures the magnitude of the error — the distance between prediction and reality — regardless of direction.

For instance, if a model predicts a value of 2 when the true value is 5, what matters isn't that the error is -3, but that the prediction is 3 units away from the actual value. To emphasize this, most loss functions eliminate the sign of the error.

The most common techniques to achieve this are:

- Taking the absolute value of the difference between prediction and actual value.
- Squaring the difference, which also penalizes larger errors more heavily.

Both approaches ensure that the model's learning focuses on reducing the size of the error, not its direction.

6.2 Types of Loss in Linear Regression

In linear regression, there are four main types of loss, which are outlined in the following table:

Loss Type	Definition	Equation
L1 Loss	The sum of the absolute values of the difference between the predicted values and the actual values.	$L1 = \sum \lvert y_i - \hat{y}_i \rvert$
Mean Absolute Error (MAE)	The average of L1 losses across a set of examples.	$MAE = \frac{1}{n} \sum \lvert y_i - \hat{y}_i \rvert$
L2 Loss	The sum of the squared differences between the predicted values and the actual values.	$L2 = \sum (y_i - \hat{y}_i)^2$
Mean Squared Error (MSE)	The average of L2 losses across a set of examples.	$MSE = \frac{1}{n} \sum (y_i - \hat{y}_i)^2$

Table 1: Common loss functions used in linear regression

The functional difference between **L1 loss** and **L2 loss** (or between **MAE** and **MSE**) is **squaring**:

- When the difference between the prediction and label is **large**, squaring makes the loss **even larger**.
- When the difference is **small** (less than 1), squaring makes the loss **even smaller**.

Tip: When processing multiple examples, it is recommended to **average the losses** across all examples—whether using MAE or MSE.

Numerical Example: Calculating Loss Functions

Using the example dataset from Section 1.2.1, consider the following data point:

- Height: 170 cm
- Age: 28 years
- Shoe Size: 42 EU
- Actual Weight: 65 kg

Recall the model parameters:

$$\hat{y} = 0.393 \times \text{Height} + 0.143 \times \text{Age} + 0.882 \times \text{ShoeSize} - 58.492$$

Calculate the predicted weight:

$$\hat{y} = 0.393 \times 170 + 0.143 \times 28 + 0.882 \times 42 - 58.492 = 49.366 \text{ kg}$$

Step 1: Calculate the error

$$\text{Error} = y - \hat{y} = 65 - 49.366 = 15.634$$

Step 2: Calculate the L1 loss (absolute error)

$$L1 = |y - \hat{y}| = |15.634| = 15.634$$

Step 3: Calculate the L2 loss (squared error)

$$L2 = (y - \hat{y})^2 = (15.634)^2 = 244.42$$

Interpretation: The L1 loss shows the magnitude of the error directly (15.634 kg), while the L2 loss penalizes larger errors more strongly (244.42).

6.3 Choosing Between MAE and MSE

When selecting a loss function for regression tasks, it's important to understand that **sensitivity to outliers** is a key—but not the only—factor influencing the choice between **Mean Absolute Error (MAE)** and **Mean Squared Error (MSE)**.

Outliers:

- **MSE** penalizes large errors more heavily by squaring the differences. This makes it

very sensitive to outliers and encourages the model to fit these extreme values more closely.

- **MAE**, on the other hand, treats all errors linearly, making it more robust to outliers.

Mathematical Properties:

- **MSE** is **differentiable everywhere**, making it easier and faster to optimize using gradient-based methods like stochastic gradient descent.
- **MAE** is **not differentiable at zero**, which can make optimization less stable and slower.

Interpretability:

- **MAE** is easier to interpret—it directly tells you the average error magnitude.
- **MSE** can be harder to interpret due to squaring the errors, but it emphasizes larger mistakes.

Data Distribution:

- If the errors in your data are **normally distributed**, **MSE** is statistically optimal.
- For data with **heavy tails or skewed error distributions**, **MAE** may yield better generalization.

Use Case Summary:

Criterion	**Prefer MSE**	**Prefer MAE**
Handle Outliers	High penalty	Robust
Optimization Simplicity	Smooth gradients	Non-differentiable at 0
Interpretability	Less intuitive (squared units)	Direct average error
Normal Error Distribution	Good fit	May underfit
Heavy-Tailed Errors	Sensitive	More stable

In practice, it is often helpful to **experiment with both loss functions** during model development and use validation metrics to determine which performs best on your specific dataset.

6.4 Linear Regression: Gradient Descent

Gradient Descent Explained with Height vs. Weight Gradient descent is an optimization technique that adjusts model parameters (weight and bias) to minimize the **loss function**—commonly the Mean Squared Error (MSE) in regression problems.

In this case, we want to train a linear model to predict a person's **weight (kg)** based on their **height (cm)**:

$$\hat{y} = w \cdot x + b$$

Where:

- x: Height (input)
- $\hat{y}$: Predicted weight
- w: Weight (slope of the line)
- b: Bias (intercept)

Step-by-Step Process

1. **Initialize parameters**
   Set $w = 0$, $b = 0$ (or small random values)
2. **Compute predictions**

$$\hat{y}_i = w \cdot x_i + b$$

3. **Calculate the loss**
   Use the Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

4. **Compute gradients**
   To minimize the loss, calculate the gradients:

$$\frac{\partial \text{MSE}}{\partial w} = \frac{2}{n} \sum_{i=1}^{n} x_i (\hat{y}_i - y_i)$$

$$\frac{\partial \text{MSE}}{\partial b} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)$$

5. **Update parameters**
   Move in the direction opposite the gradient (learning rate α):

$$w := w - \alpha \cdot \frac{\partial \text{MSE}}{\partial w}$$

$$b := b - \alpha \cdot \frac{\partial \text{MSE}}{\partial b}$$

- A small α leads to slow but stable convergence.
- A large α can cause the model to overshoot and fail to converge.

Repeat steps 2–5 for several iterations until the loss converges (stops changing significantly).

As gradient descent progresses, each update to the weight and bias reduces the loss, bringing the model closer to an optimal fit. In our example, we ran the process for six iterations, and each step brought noticeable improvements.

In real-world scenarios, training typically continues until the model reaches a **stable state**, known as **convergence**. At this point, further updates to the parameters no longer lead to meaningful reductions in loss.

After convergence, the algorithm may still make tiny adjustments, causing the loss to **hover slightly** around a minimum value. This behavior is expected and indicates that the model has settled into a configuration that minimizes prediction error.

Key Point: You can usually tell the model has converged when the loss plateaus and no longer shows significant change with further training steps.

Algorithm 1 Gradient Descent for Linear Regression

1: Initialize $w \leftarrow 0$, $b \leftarrow 0$
2: Choose learning rate α and number of iterations T
3: **for** $t = 1$ to T **do**
4: Compute predictions: $\hat{y}_i = w \cdot x_i + b$ for all i
5: Compute gradients:

$$\frac{\partial \text{MSE}}{\partial w} = \frac{2}{n} \sum_{i=1}^{n} x_i(\hat{y}_i - y_i)$$

$$\frac{\partial \text{MSE}}{\partial b} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)$$

6: Update parameters:

$$w \leftarrow w - \alpha \cdot \frac{\partial \text{MSE}}{\partial w}$$

$$b \leftarrow b - \alpha \cdot \frac{\partial \text{MSE}}{\partial b}$$

7: **end for**=0

Numerical Application of Gradient Descent

Consider the dataset:

Height (cm)	**Weight (kg)**
150	50
160	55
170	65
180	72
190	80

Parameters Initialization:

$$w = 0, \quad b = 0, \quad \alpha = 0.0001, \quad n = 5$$

Iteration 0:

- **Predictions:**

$$\hat{y}_i = wx_i + b = 0$$

- **Errors:**

$$\hat{y}_i - y_i = -y_i$$

$$-50, \quad -55, \quad -65, \quad -72, \quad -80$$

- **Compute gradients:**

$$\frac{\partial \text{MSE}}{\partial w} = \frac{2}{n}\sum_{i=1}^{n} x_i(\hat{y}_i - y_i) = \frac{2}{5}(150\times-50+160\times-55+170\times-65+180\times-72+190\times-80)$$

Calculate the sum:

$$-7500 - 8800 - 11050 - 12960 - 15200 = -55510$$

Thus,

$$\frac{\partial \text{MSE}}{\partial w} = \frac{2}{5} \times (-55510) = -22204$$

$$\frac{\partial \text{MSE}}{\partial b} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) = \frac{2}{5}(-50 - 55 - 65 - 72 - 80) = \frac{2}{5} \times (-322) = -128.8$$

- **Update parameters:**

$$w := w - \alpha \cdot \frac{\partial \text{MSE}}{\partial w} = 0 - 0.0001 \times (-22204) = 2.2204$$

$$b := b - \alpha \cdot \frac{\partial \text{MSE}}{\partial b} = 0 - 0.0001 \times (-128.8) = 0.01288$$

New predictions:

$$\hat{y}_i = 2.2204 \times x_i + 0.01288$$

Height x_i	Actual y_i	Prediction $\hat{y}_i$	Error $\hat{y}_i - y_i$
150	50	333.06	+283.06
160	55	355.27	+300.27
170	65	377.48	+312.48
180	72	399.69	+327.69
190	80	421.90	+341.90

Note: The errors increased, indicating that the learning rate may be too large causing overshoot. In practice, reduce α or use smaller steps for stable convergence.

—

Summary Table:

Iteration	w	b	Gradient w	Gradient b
0	0	0	-22204	-128.8
1	2.2204	0.01288	Recalculate	Recalculate

You can continue updating parameters iteratively using the same steps until the loss converges.

Visualization of Gradient Descent

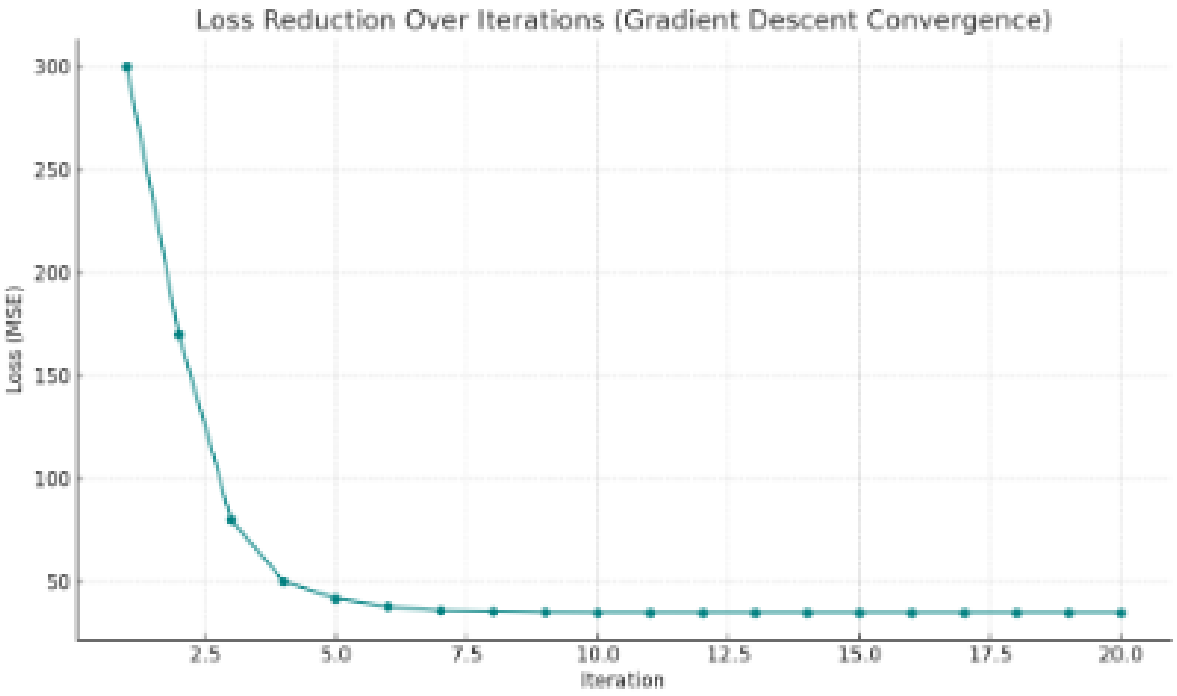


Figure 4: Loss function surface showing convergence as weight (w) and bias (b) are updated

The figure above illustrates how the prediction line gradually improves as the model updates w and b over time, reducing the loss.

Notes:

- A good learning rate is crucial:
  - Too high $\Rightarrow$ the model may overshoot and diverge.
  - Too low $\Rightarrow$ the convergence becomes slow and inefficient.
- Gradient descent generalizes to:

- Higher-dimensional data (multiple features),
- More complex models (e.g., neural networks),
- Various types of loss functions.

Convergence and Convex Functions

In a linear regression model, the loss function—often the **Mean Squared Error (MSE)**—quantifies how well the model's predictions align with the actual data. When plotted in terms of the model parameters (e.g., weight w and bias b), this loss function forms a **convex surface**.

This convexity means that:

- There is a single global minimum (no local minima).
- Gradient descent can reliably converge to the optimal point.

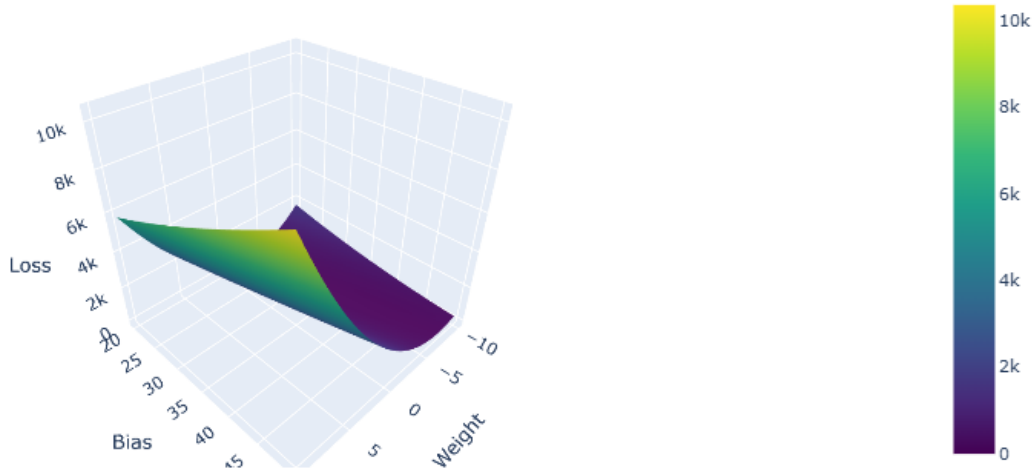


Figure 5: Convex loss surface: gradient descent rolls down to the global minimum

As shown in the figure above, the shape of the loss surface resembles a bowl. Gradient descent works by iteratively adjusting the parameters w and b, moving "downhill" on this surface. Each step brings the model closer to the lowest point—the minimum loss.

This process is analogous to a ball rolling down a hill toward the bottom:

- If the learning rate is well chosen, the ball smoothly rolls to the bottom.
- If it's too large, the ball may overshoot and oscillate or even diverge.

- If it's too small, the descent becomes very slow.

Thus, convergence behavior is strongly tied to the loss surface being convex and the learning rate being appropriately tuned.

Linear Regression: Hyperparameters

In machine learning, **hyperparameters** are the settings defined *before* training begins. They guide how the model learns. In contrast, **parameters** (like weights and biases) are learned *during* training.

Common Hyperparameters

- **Learning rate**: Controls the size of updates to model parameters during gradient descent.
- **Batch size**: Number of training examples used in one iteration to compute the gradient.
- **Epochs**: One full pass through the entire training dataset.

Learning Rate

The learning rate determines how fast or slow a model learns. It's a key factor in whether your model successfully **converges** (i.e., reaches the lowest possible loss).

- **Too low**: The model learns slowly and takes a long time to converge.
- **Too high**: The model may never converge—bouncing around or even diverging.

Update Rule: During gradient descent, the model updates parameters as follows:

$$\text{new\_weight} = \text{old\_weight} - (\text{learning\_rate} \times \text{gradient})$$

Example: If the gradient is 2.5 and the learning rate is 0.01:

$$\text{Change} = 0.01 \times 2.5 = 0.025$$

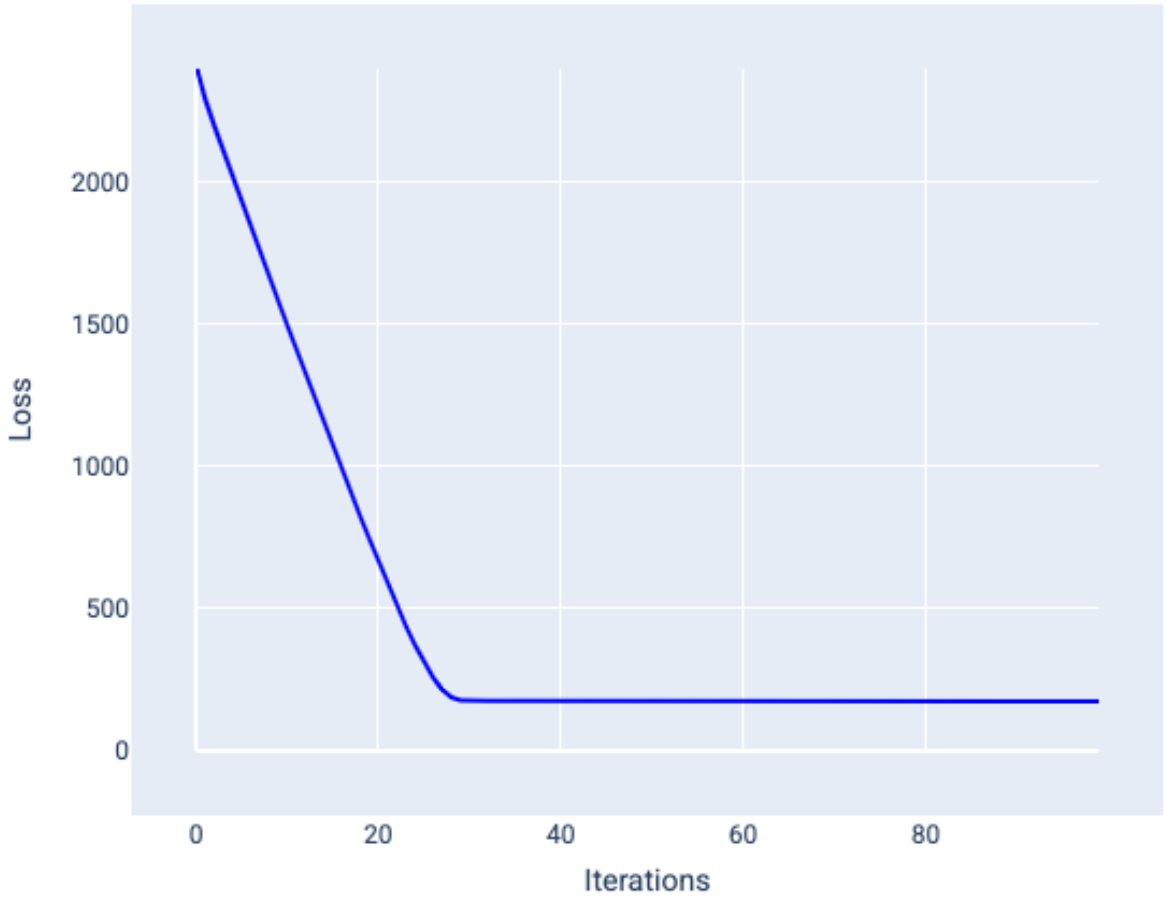


Figure 6: Loss curve with a good learning rate: quick convergence.

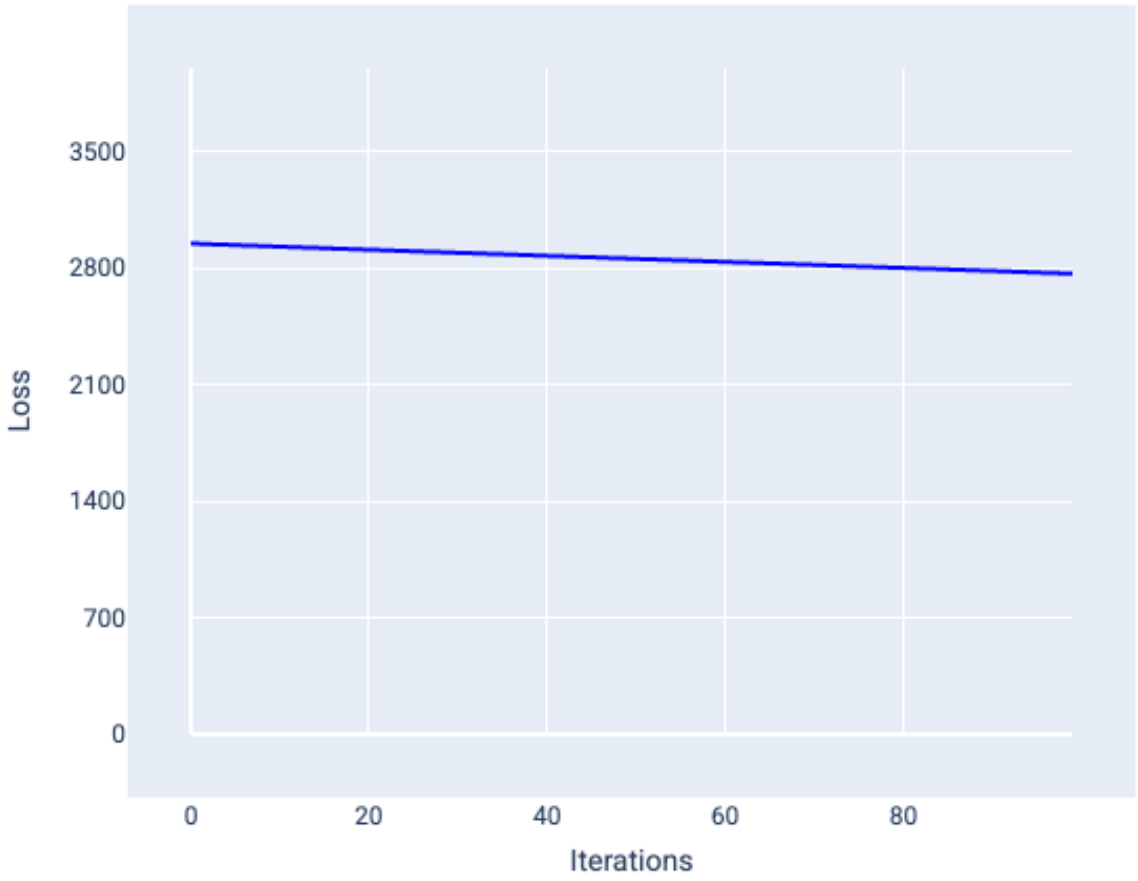


Figure 7: Loss curve with a small learning rate: slow convergence.

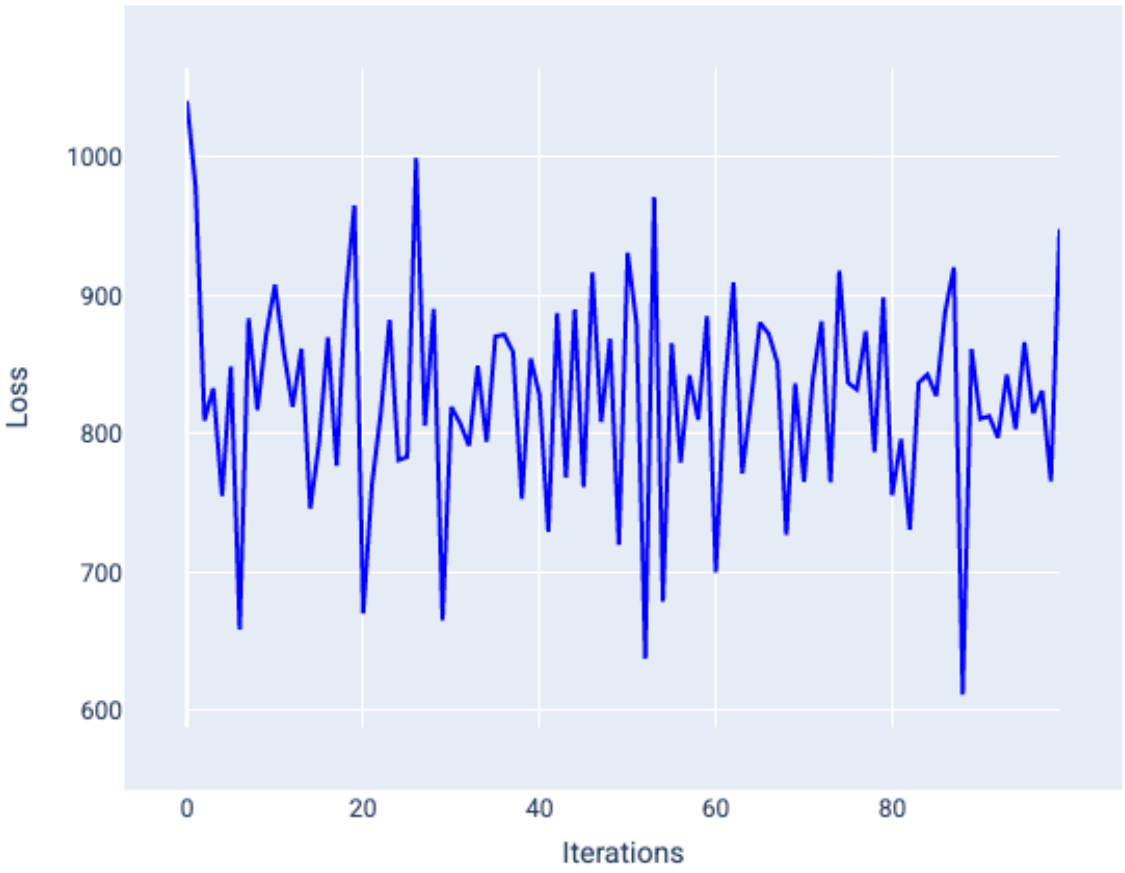


Figure 8: Loss curve with too high a learning rate: erratic convergence.

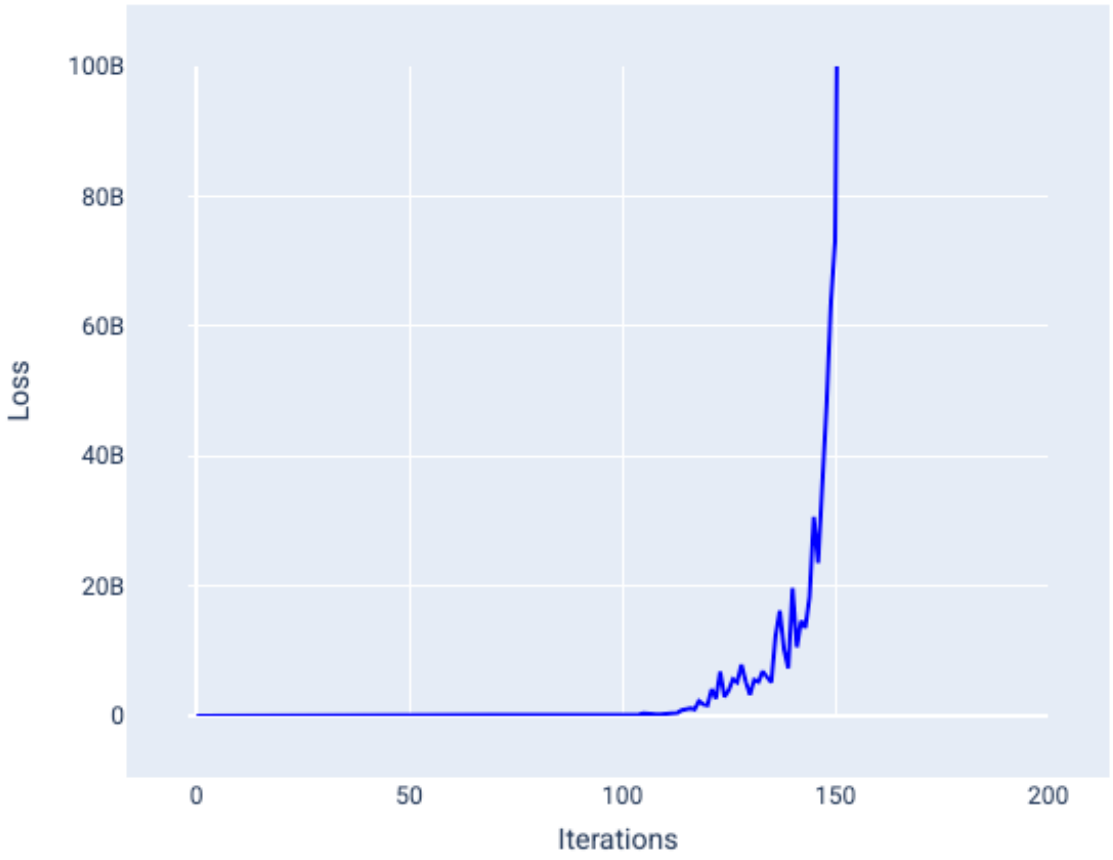


Figure 9: Loss curve diverging due to an overly large learning rate.

Batch Size

Batch size defines how many training examples are processed before updating model parameters. Since using the entire dataset can be computationally expensive, we use alternative strategies:

Stochastic Gradient Descent (SGD):

- Updates weights after each training example.
- Computationally efficient but introduces **noise** in the loss curve.

Mini-Batch SGD:

- Uses a small group (batch) of examples to compute the average gradient.

- Balances efficiency and stability, resulting in smoother convergence.

Noise and Its Role in Training: Noise may seem undesirable, but a controlled amount of noise helps the model avoid local minima and improves generalization. It:

- Prevents overfitting.
- Encourages exploration of the parameter space.
- Allows convergence to a better global solution.

Epochs in Training

An **epoch** represents a full pass through the training data. For example, with 1,000 training samples and a batch size of 100, one epoch involves 10 iterations.

- Multiple epochs are typically required for the model to learn effectively.
- The number of epochs is a tunable hyperparameter.
- More epochs generally improve performance but increase training time.

Choosing the right number of epochs often requires experimentation to find a balance between underfitting and overfitting.

6.5 Feature Mapping and Polynomial Regression

Sometimes, data does not follow a straight-line pattern. Linear regression, by itself, can only model linear relationships. But what if the underlying pattern is curved or more complex?

This is where **feature mapping** comes in. Instead of using just the original input x, we create new features from it—like x^2, x^3, $\ldots$, x^M—and use those as inputs to linear regression. This allows the model to fit more flexible, non-linear patterns.

Polynomial Feature Mapping

We transform the input into a polynomial feature vector:

$$\phi(x) = [1, x, x^2, x^3, \ldots, x^M]$$

Then we apply linear regression to this transformed input:

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M$$

This model is **non-linear in** x, but still **linear in the weights** w, so we can solve it using ordinary linear regression techniques.

Why Use Polynomial Features?

- Simple data $\Rightarrow$ Use a low-degree polynomial (e.g., $M = 1$ or $M = 3$).
- Complex patterns $\Rightarrow$ Higher-degree polynomials might fit better.

  Be careful: A degree that's too high can lead to **overfitting**, where the model memorizes the data but performs poorly on unseen examples.

6.6 Regularization

To prevent overfitting when using many features or high-degree polynomials, we apply **regularization**.

Regularization adds a penalty to the loss function to discourage the model from assigning large weights. This keeps the model simpler and improves its generalization to new data.

Types of Regularization

- **L2 Regularization (Ridge):** Adds a penalty term to the loss:

$$\text{Loss}_{\text{ridge}} = \text{Loss}_{\text{MSE}} + \lambda \sum_{i=1}^{M} w_i^2$$

- **L1 Regularization (Lasso):** Adds a penalty based on the absolute values of the weights:

$$\text{Loss}_{\text{lasso}} = \text{Loss}_{\text{MSE}} + \lambda \sum_{i=1}^{M} |w_i|$$

Key Idea: Regularization helps balance model complexity and accuracy. It is especially useful when working with polynomial features or many input variables.

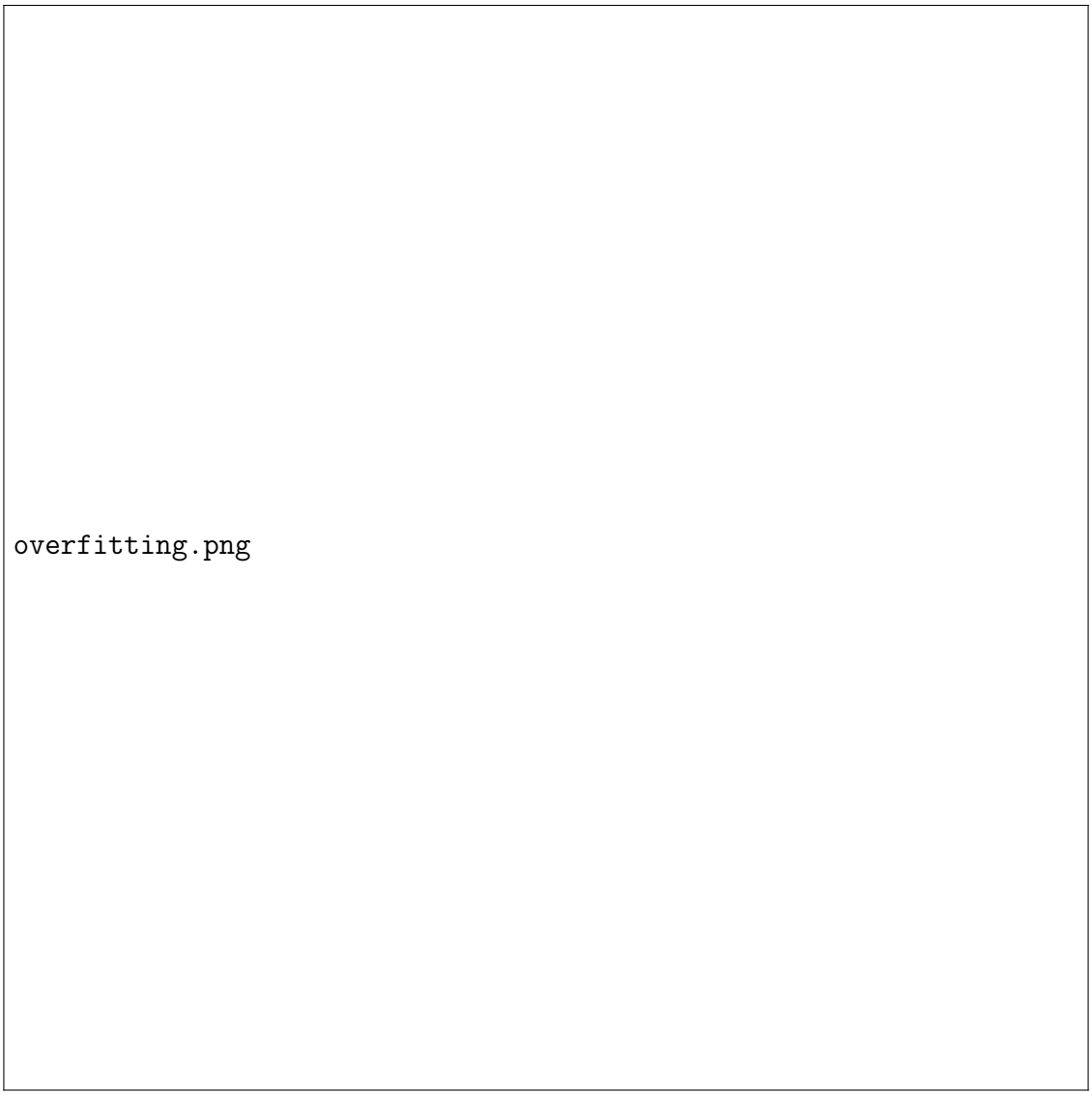


Figure 10: Model complexity vs generalization: Underfitting (left, $M = 0$), good generalization (middle, $M = 3$), and overfitting (right, $M = 9$).

7 Conclusion

Gradient descent enables linear regression to be trained efficiently on large datasets by iteratively refining the parameters. It serves as a foundation for many machine learning methods, including deep learning.