Deep Neural Network

Module # 3 : Linear Neural Networks for Regression

Seetha Parameswaran

BITS Pilani WILP

The instructor is gratefully acknowledging the authors who made their course materials freely available online.
This deck is prepared by Seetha Parameswaran.

Table of Contents

1. Regression
2. Linear Regression
3. The Linear Model: Single Neuron
4. Worked Example
5. Evaluation
6. Python Implementation
7. Implementation Tips
8. Summary

What is Regression?

Definition

Regression is a supervised learning task that predicts a continuous-valued output based on input features.

Key Characteristics:

- Input: Feature vector $\mathbf{x} \in \mathbb{R}^d$
- Output: Real-valued target $y \in \mathbb{R}$
- Goal: Learn a function $f : \mathbb{R}^d \to \mathbb{R}$ such that $y \approx f(\mathbf{x})$

Examples of Regression Problems

1. House Price Prediction
   - Input: Size, location, number of rooms
   - Output: Price in dollars
2. Temperature Forecasting
   - Input: Historical weather data, season, humidity
   - Output: Temperature in degrees
3. Stock Price Prediction
   - Input: Historical prices, trading volume, market indicators
   - Output: Future stock price

Table of Contents

1. Regression
2. Linear Regression
3. The Linear Model: Single Neuron
4. Worked Example
5. Evaluation
6. Python Implementation
7. Implementation Tips
8. Summary

What is Linear Regression?

Definition

Linear regression assumes the output is a linear combination of the input features.

Mathematical Form:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d \tag{1}$$

where:

- $\mathbf{x} = [x_1, x_2, \ldots, x_d]^T$ are the input features
- $\mathbf{w} = [w_1, w_2, \ldots, w_d]^T$ are the weights
- w_0 is the bias term

Examples of Linear Regression

1. Salary Prediction
   - $\text{Salary} = w_0 + w_1 \times \text{Years of Experience}$
2. Crop Yield Prediction
   - $\text{Yield} = w_0 + w_1 \times \text{Rainfall} + w_2 \times \text{Fertilizer}$
3. Energy Consumption
   - $\text{Energy} = w_0 + w_1 \times \text{Temperature} + w_2 \times \text{Hour} + w_3 \times \text{Day}$

The relationship between inputs and output is assumed to be linear.

Visual Example: 2D Linear Regression

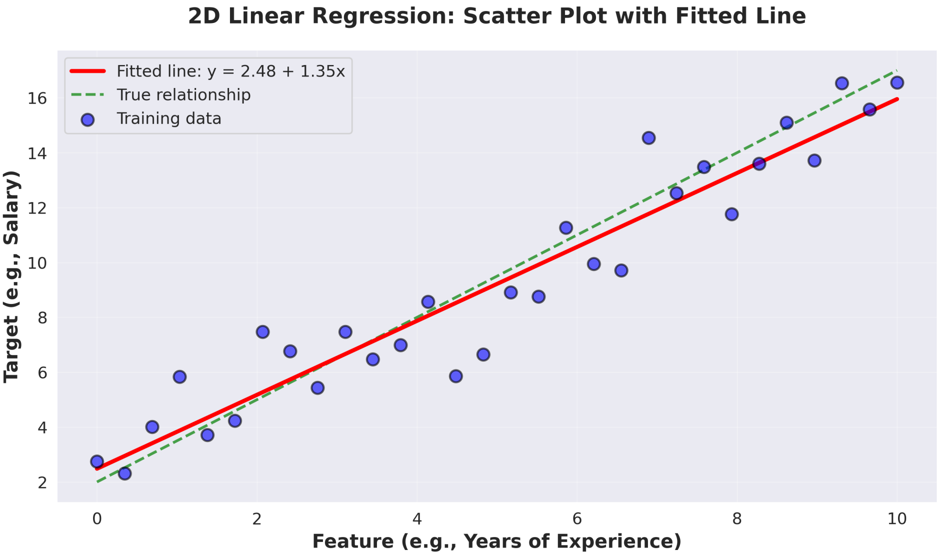


Table of Contents

1. Regression
2. Linear Regression
3. The Linear Model: Single Neuron
4. Worked Example
5. Evaluation
6. Python Implementation
7. Implementation Tips
8. Summary

The Four Components

A complete machine learning system consists of:

1. Data: d-dimensional input vectors and target outputs
2. Model: Single neuron implementing linear function
3. Objective Function: Measures prediction error
4. Learning Algorithm: Gradient descent to optimize weights

Matrix Form of Data

For efficient computation, we organize data into matrices:

Input Data = Augmented Features:

Add bias term: $\tilde{\mathbf{x}}^{(i)} = [1, x_1^{(i)}, \ldots, x_d^{(i)}]^T$

Each row = one training example

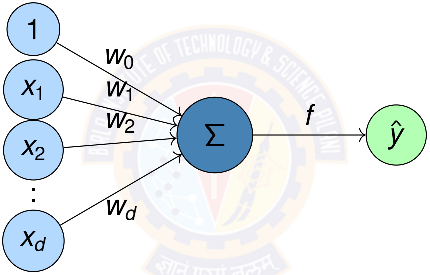
Design Matrix: $\mathbf{X} \in \mathbb{R}^{N\times(d+1)}$ $\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_d^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(N)} & \cdots & x_d^{(N)} \end{bmatrix}$

Target Vector: $\mathbf{y} \in \mathbb{R}^N$ $\mathbf{y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \ldots & y^{(N)} \end{bmatrix}^\top$

Weight Vector: $\mathbf{w} \in \mathbb{R}^{d+1}$ $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 & \ldots & w_d \end{bmatrix}^\top$

Linear Model as a Single Neuron

Model linear regression as a single artificial neuron:



- Input: d-dimensional feature vector $\mathbf{x} \in \mathbb{R}^d$
- Output: Single predicted value $\hat{y} \in \mathbb{R}$

Linear Model Prediction

Model: A single neuron computes a weighted sum and applied identity activation function.

Prediction for input $\mathbf{x}$:

$$\hat{y} = f(\mathbf{w}^T\mathbf{x}) = f(w_0 + w_1x_1 + w_2x_2 + \cdots + w_dx_d) \tag{2}$$

where:

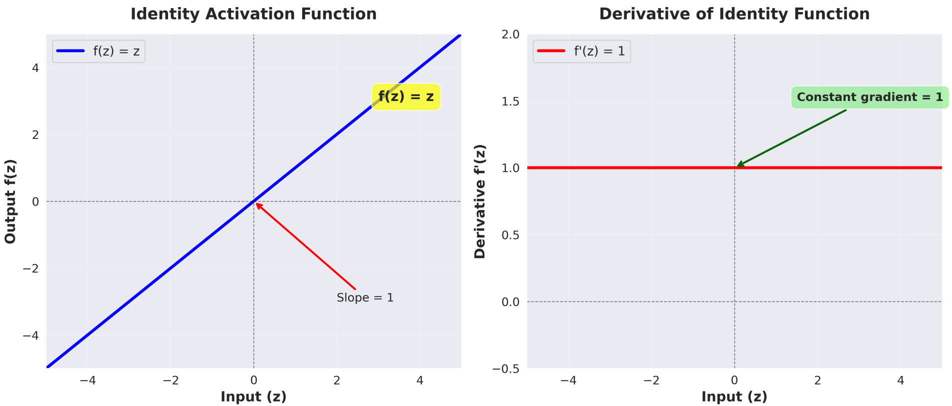
- $\mathbf{w} = [w_0, w_1, \ldots, w_d]^T \in \mathbb{R}^{d+1}$ are the model parameters
- $\mathbf{x} = [1, x_1, \ldots, x_d]^T \in \mathbb{R}^{d+1}$ is the input
- $f(\cdot)$ is the identity activation $f(z) = z$

Vectorized form: for all predictions

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} \tag{3}$$

Identity Activation Function

$$f(z) = z \tag{4}$$



Identity Activation Function

Properties of Identity Function

- Output: Continuous, $(-\infty, \infty)$
- Differentiable everywhere: $f'(z) = 1$
- Allows output to take any real value
- Gradient flows unchanged through the neuron
- Perfect for predicting continuous targets

Why identity for regression?

- Target values are continuous (prices, temperatures, etc.)
- Need to predict any real number, not just binary $\{ 0,1 \}$
- Differentiability enables gradient-based optimization
- No information loss from input to output

Objective Function: Squared Error Loss

We need to measure how well our model fits the data.

Loss for single example:

$$\ell\left(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)}\right) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{2}\left(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)}\right)^2 \tag{5}$$

Total Loss (Mean Squared Error):

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)}\right)^2 \tag{6}$$

Goal: Find $\mathbf{w}^*$ that minimizes $J(\mathbf{w})$

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w}) \tag{7}$$

Why Squared Error?

Advantages of squared error loss:

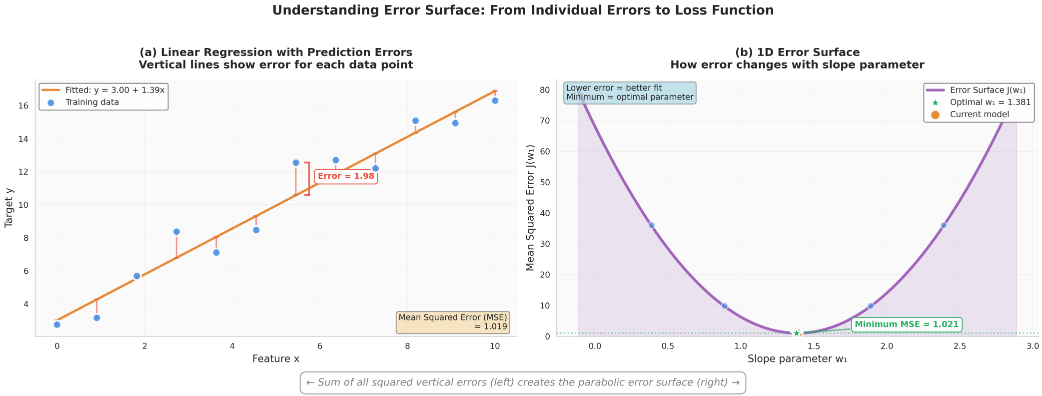
- Differentiable: Smooth everywhere, easy to optimize
- Convex: Single global minimum (for linear models)
- Penalizes large errors: Quadratic growth
- Statistical interpretation: Maximum likelihood under Gaussian noise

Vector Form:

$$J(\mathbf{w}) = \frac{1}{2N}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 = \frac{1}{2N}(\mathbf{X}\mathbf{w} - \mathbf{y})^T(\mathbf{X}\mathbf{w} - \mathbf{y}) \tag{8}$$

where $\mathbf{X} \in \mathbb{R}^{N\times(d+1)}$ is the design matrix and $\mathbf{y} \in \mathbb{R}^N$ is the target vector.

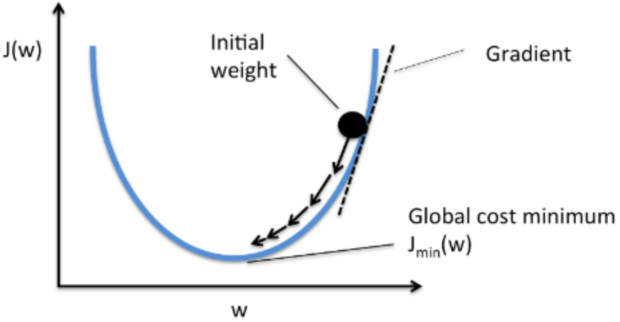
Visualizing the Error Surface



The error surface is convex with a single global minimum.

Gradient Descent Algorithm

Idea: Iteratively update weights in the direction of steepest descent



where $\eta > 0$ is the learning rate (step size).

Intuition: Move downhill on the error surface to find the minimum.

Batch Gradient Descent Algorithm

Algorithm 1: Gradient Descent for Linear Regression

Input: Dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, learning rate η, tolerance ϵ

Output: Learned weights $\mathbf{w}$

```
Initialize w^(0) = 0 (or random);
t ← 0;
while not converged do
    Compute gradient: ∇J(w^(t)) = (1/N) X^T (X w^(t) − y);
    Update weights: w^(t+1) = w^(t) − η ∇J(w^(t));
    if ||∇J(w^(t+1))|| < ε then
        break // Converged
    t ← t + 1;
return w^(t+1);
```

Computing the Gradient

Gradient of squared error loss:

$$\nabla J(\mathbf{w}) = \frac{\partial J}{\partial \mathbf{w}} = \frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)}\right)\mathbf{x}^{(i)} \tag{9}$$

Matrix Form:

$$\nabla J(\mathbf{w}) = \frac{1}{N}\mathbf{X}^T\left(\mathbf{X}\mathbf{w} - \mathbf{y}\right) \tag{10}$$

where $\mathbf{X}^T \in \mathbb{R}^{(d+1)\times N}$, $\mathbf{X}\mathbf{w} - \mathbf{y} \in \mathbb{R}^N$, so $\nabla J \in \mathbb{R}^{d+1}$

Gradient Descent Update Rule

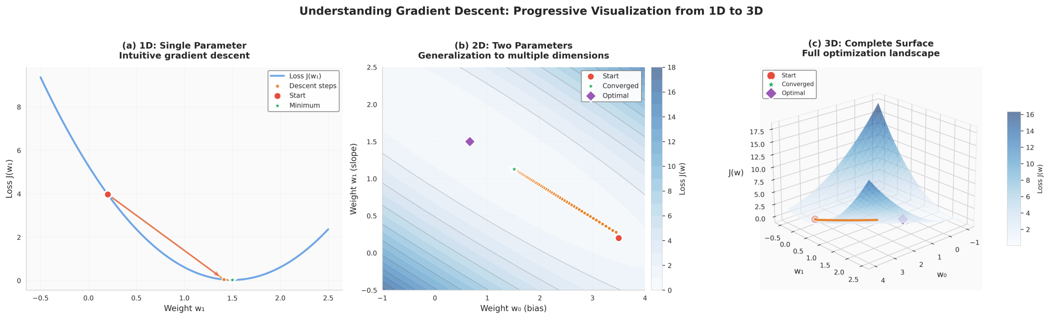
Update equation:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \frac{\eta}{N}\mathbf{X}^T\left(\mathbf{X}\mathbf{w}^{(t)} - \mathbf{y}\right) \tag{11}$$

Key Parameters:

- Learning rate η: Controls step size
  - Too large: May overshoot minimum
  - Too small: Slow convergence
- Stopping criterion: $\|\nabla J(\mathbf{w})\| < \epsilon$ or max iterations

Visualizing Gradient Descent



Each step moves perpendicular to contour lines toward the minimum.

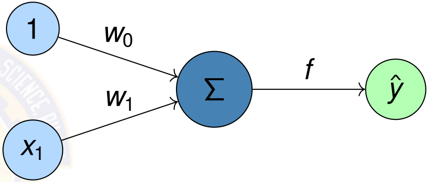
Table of Contents

1. Regression
2. Linear Regression
3. The Linear Model: Single Neuron
4. **Worked Example**
5. Evaluation
6. Python Implementation
7. Implementation Tips
8. Summary

Numerical Example: Setup

Tiny Dataset: Predict house price from size

Size (x_1)	Price (y)
1	2
2	4
3	5



Matrix Formulation:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix} \in \mathbb{R}^{3\times 2}, \quad \mathbf{y} = \begin{bmatrix} 2 \\ 4 \\ 5 \end{bmatrix} \in \mathbb{R}^{3}, \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \in \mathbb{R}^{2}$$

Numerical Example: Initialization

Hyperparameters: $\eta = 0.1 \quad N = 3$

Initialize weights: $\mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Vectorized Prediction: $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$

Initial predictions: $\hat{\mathbf{y}}^{(0)} = \mathbf{X}\mathbf{w}^{(0)} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$

Loss Function: $J(\mathbf{w}) = \frac{1}{2N} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$

Initial loss: $J(\mathbf{w}^{(0)}) = \frac{1}{6} \left\| \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ 4 \\ 5 \end{bmatrix} \right\|^2 = \frac{1}{6}(4 + 16 + 25) = 7.5$

Numerical Example: Computing Gradient

Error vector: $\mathbf{e}^{(0)} = \hat{\mathbf{y}}^{(0)} - \mathbf{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} -2 \\ -4 \\ -5 \end{bmatrix}$

Gradient computation: $\nabla J(\mathbf{w}^{(0)}) = \frac{1}{N}\mathbf{X}^T\mathbf{e}^{(0)}$

$$\nabla J(\mathbf{w}^{(0)}) = \frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix}\begin{bmatrix} -2 \\ -4 \\ -5 \end{bmatrix}$$

$$= \frac{1}{3}\begin{bmatrix} -2 - 4 - 5 \\ -2 - 8 - 15 \end{bmatrix} = \begin{bmatrix} -3.67 \\ -8.33 \end{bmatrix}$$

Numerical Example: Weight Update

Gradient descent update: $\mathbf{w}^{(1)} = \mathbf{w}^{(0)} - \eta \nabla J(\mathbf{w}^{(0)})$

Computing new weights: $\mathbf{w}^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - 0.1 \begin{bmatrix} -3.67 \\ -8.33 \end{bmatrix} = \begin{bmatrix} 0.367 \\ 0.833 \end{bmatrix}$

New predictions: $\hat{\mathbf{y}}^{(1)} = \mathbf{X}\mathbf{w}^{(1)}$

$$\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 0.367 \\ 0.833 \end{bmatrix} = \begin{bmatrix} 1.20 \\ 2.03 \\ 2.87 \end{bmatrix}$$

Numerical Example: New Loss

New error vector: $\mathbf{e}^{(1)} = \hat{\mathbf{y}}^{(1)} - \mathbf{y} = \begin{bmatrix} 1.20 \\ 2.03 \\ 2.87 \end{bmatrix} - \begin{bmatrix} 2 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} -0.80 \\ -1.97 \\ -2.13 \end{bmatrix}$

Loss after one iteration: $J(\mathbf{w}^{(1)}) = \frac{1}{6}\|\mathbf{e}^{(1)}\|^2 = \frac{1}{6}(0.64 + 3.88 + 4.54) \approx 1.51$

Progress:

- Initial loss: $J(\mathbf{w}^{(0)}) = 7.5$
- After 1 iteration: $J(\mathbf{w}^{(1)}) = 1.51$
- **80% reduction!** $\checkmark$

Continue until $\|\nabla J(\mathbf{w})\| < \epsilon$ or max iterations reached

Computational Graph for One Gradient

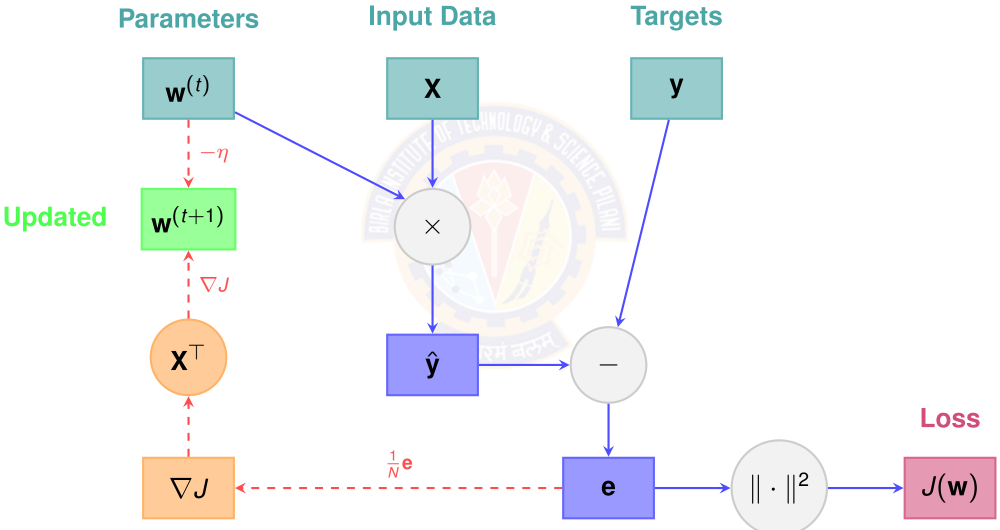


Table of Contents

1. Regression
2. Linear Regression
3. The Linear Model: Single Neuron
4. Worked Example
5. Evaluation
6. Python Implementation
7. Implementation Tips
8. Summary

Model Evaluation: Training vs Test Error

Key Principle: Evaluate on unseen data

Data Split:

- Training set: Used to learn weights **w** (typically 90-99%)
- Test set: Used to evaluate generalization (typically 1-10%)

$$\text{Training Error:} \quad J_{\text{train}} = \frac{1}{N_{\text{train}}} \sum_{i \in \text{train}} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

$$\text{Test Error:} \quad J_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{i \in \text{test}} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

Goal: Minimize test error (generalization performance)

Evaluation Metrics

1. Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 \tag{12}$$

2. Root Mean Squared Error (RMSE):

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2} \tag{13}$$

3. Mean Absolute Error (MAE):(Less sensitive to outliers than MSE)

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}\mid y^{(i)} - \hat{y}^{(i)} \mid \tag{14}$$

R^2 Score (Coefficient of Determination)

Definition: Proportion of variance explained by the model

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{N}(y^{(i)} - \bar{y})^2} = 1 - \frac{\text{SS}_{\text{residual}}}{\text{SS}_{\text{total}}} \tag{15}$$

where $\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y^{(i)}$ is the mean.

Interpretation:

- $R^2 = 1$: Perfect fit (all variance explained)
- $R^2 = 0$: Model no better than predicting mean
- $R^2 < 0$: Model worse than baseline (possible on test set)
- Typical: $0.7 < R^2 < 0.9$ indicates good fit

Table of Contents

1. Regression
2. Linear Regression
3. The Linear Model: Single Neuron
4. Worked Example
5. Evaluation
6. Python Implementation
7. Implementation Tips
8. Summary

Perceptron Demo Python Code

- `https://colab.research.google.com/drive/10whoZYodz1H02mbVwsmTudwxaU18ukaI?usp=sharing`

Student pl note:
The Python notebook is shared for anyone who has access to the link and the access is restricted to use BITS email id. So please do not access from non-BITS email id and send requests for access. Access for non-BITS email id will NOT be granted.

Table of Contents

1. Regression
2. Linear Regression
3. The Linear Model: Single Neuron
4. Worked Example
5. Evaluation
6. Python Implementation
7. Implementation Tips
8. Summary

Tips for Implementation

1. Choosing Learning Rate η:
   - Start with $\eta \in \{0.001, 0.01, 0.1, 1.0\}$
   - Plot loss vs iterations for each
   - If loss increases or becomes NaN $\rightarrow$ decrease η
   - If convergence too slow $\rightarrow$ increase η
   - Use learning rate schedules for fine-tuning
2. When to Stop Training:
   - Gradient magnitude: $\|\nabla J\| < \epsilon$ (e.g., $\epsilon = 10^{-4}$)
   - Loss change: $|J^{(t)} - J^{(t-1)}| < \epsilon$
   - Maximum iterations reached (e.g., $T = 1000$ or 10,000)
   - Validation loss starts increasing (early stopping for overfitting)

More Practical Tips

3. Weight Initialization:
   - Zero initialization: $\mathbf{w}^{(0)} = \mathbf{0}$ (works well for linear regression)
   - Avoid large initial values (can cause divergence or overflow)
4. Feature Engineering:
   - Domain knowledge: Create meaningful derived features (e.g., price per sq ft)
   - Feature selection: Remove irrelevant/redundant features
5. Feature Scaling and Normalization:
   - Features with different scales can slow convergence
   - 1. Min-Max Scaling: Scale to $[0, 1]$ $x_j' = \frac{x_j - \min(x_j)}{\max(x_j) - \min(x_j)}$
   - 2. Standardization (Z-score): Zero mean, unit variance $x_j' = \frac{x_j - \mu_j}{\sigma_j}$

Debugging Checklist

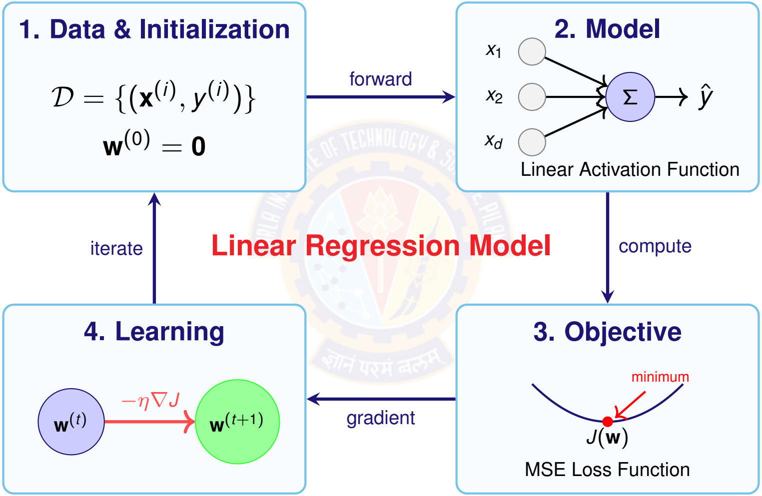
Common Issues and Solutions:

- Loss is NaN/Inf:
  - Learning rate too large
  - Numerical overflow (check feature scales)
  - Solution: Decrease η, scale features
- Training works but test error high:
  - Overfitting (too many features relative to data)
  - Solution: Regularization, more data, or simpler model
- Loss oscillating:
  - Learning rate too large
  - Solution: Decrease η or use learning rate decay
- Loss not decreasing:
  - Learning rate too small
  - Bug in gradient computation (check signs!)
  - Wrong sign in update rule (should be minus)
  - Solution: Increase η, verify code

Table of Contents

1 Regression
2 Linear Regression
3 The Linear Model: Single Neuron
4 Worked Example
5 Evaluation
6 Python Implementation
7 Implementation Tips
8 Summary

Summary: Putting It All Together



This framework extends to more complex models: neural networks, deep learning!

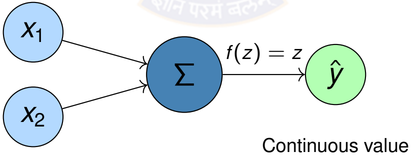
Key Takeaways

- Regression predicts continuous outputs from input features.
- Linear regression assumes a linear relationship: $y = \mathbf{w}^T\mathbf{x} + w_0$
- A single neuron with identity activation implements linear regression.
- Squared error loss provides a convex objective function.
- Gradient descent iteratively optimizes weights: $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J$
- Feature scaling is critical for fast convergence.
- Evaluation on test set ensures generalization.

Linear Regression

Key insight: Linear relationship between inputs and output, optimized by minimizing squared errors.

- Problem: Predict continuous values: $y \in \mathbb{R}$
- Model: Single neuron with identity activation
- Output: Predicted value $\hat{y} = \mathbf{w}^T\mathbf{x} \in (-\infty, \infty)$
- Loss: Mean Squared Error (MSE)
- Training: Batch Gradient Descent (GD)



References

- Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2023). *Dive into Deep Learning*. Cambridge University Press. Chapter 3 (Linear Neural Networks for Regression). https://d2l.ai/
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Chapter 5 (Machine Learning Basics). https://www.deeplearningbook.org/
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. Chapter 3 (Linear Models for Regression).
- Murphy, K. P. (2022). *Probabilistic Machine Learning: An Introduction*. MIT Press. Chapter 11 (Linear Regression). https://probml.github.io/pml-book/

Thank You!