Fondamental of Data Science, 1st semester at Sapienza, Pt 2

Emanuele Iaccarino

October 2024

Chapter 1

Linear Regression

1.1 Introduction to Linear Regression

Linear regression is a supervised learning algorithm used for predicting real-valued outputs. It models the relationship between one or more input features and a target variable using a linear function.

1.1.1 Univariate Linear Regression

In the case of univariate linear regression, the model involves one feature x and predicts the output y using the hypothesis:

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1.1}$$

where θ_0 is the intercept and θ_1 is the slope of the line. The goal is to choose parameters θ_0 and θ_1 such that the hypothesis fits the training data well.

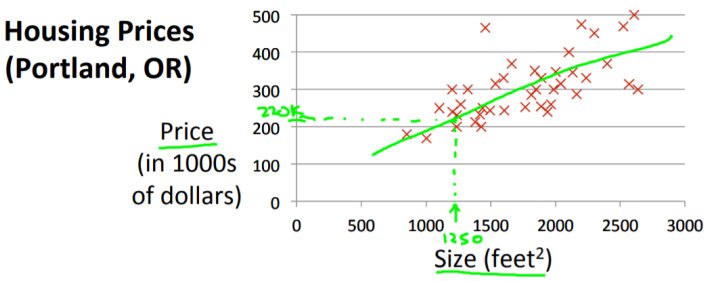


Figure 1.1: Example of Univariate Linear Regression

The **Training set** (A set of training examples used to learn the model) consists of m samples (x_i, y_i), which are used to learn the parameters. The dataset can be represented as:

$$S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\} \tag{1.2}$$

The objective of the **Learning Algorithm** is to find a function $h(x)$ that approximates the target value y (e.g., the price of a house) as closely as possible, using input features such as house size and other characteristics..

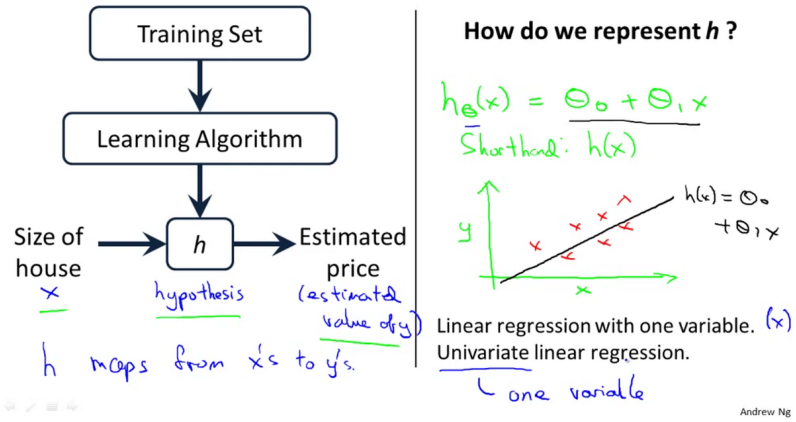


Figure 1.2: Training set Example

Assumptions Linear regression assumes that the joint probability distribution $p(X, Y)$ is the same for both the training and test datasets. The training examples are assumed to be identically distributed and independently distributed (i.i.d).

1.1.2 Regression Function

The regression function for univariate linear regression is given by:

$$f_\theta(x) = \theta_0 + \theta_1 x \tag{1.3}$$

where θ_0 and θ_1 are parameters to be learned from the training data.

For convenience, the input feature vector x_i can be represented as:

$$x_i = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \in \mathbb{R}^m, \quad y_i \in \mathbb{R} \tag{1.4}$$

1.1.3 Multivariate Linear Regression

For multivariate linear regression, multiple features are used. The hypothesis function is:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = \theta^T x \tag{1.5}$$

where x is the feature vector $[x_0, x_1, \ldots, x_n]^T$ with $x_0 = 1$ (for convenience of notation) and θ is the parameter vector.

Key Assumptions:

- **Linearity:** The relationship between x and y is linear.
- **Independence:** The data samples are independent of each other.
- **Homoscedasticity:** The variance of errors remains constant across the range of input features.
- **Normality:** The residuals (differences between actual and predicted y) are normally distributed.

1.2 Cost Function

The cost function measures how well the hypothesis fits the training data. The goal of linear regression is to minimize this cost function, which is defined as:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{1.6}$$

where m is the number of training examples, $x^{(i)}$ is the i-th training example, and $y^{(i)}$ is the corresponding target value.

The cost function represents the **Mean Squared Error (MSE)** between the predicted values and the actual target values.

1.2.1 Visualizing the Cost Function

The cost function $J(\theta_0, \theta_1)$ can be visualized as a 3D surface plot or a contour plot. These visualizations help in understanding how the cost varies with different parameter values and how gradient descent converges to the minimum.

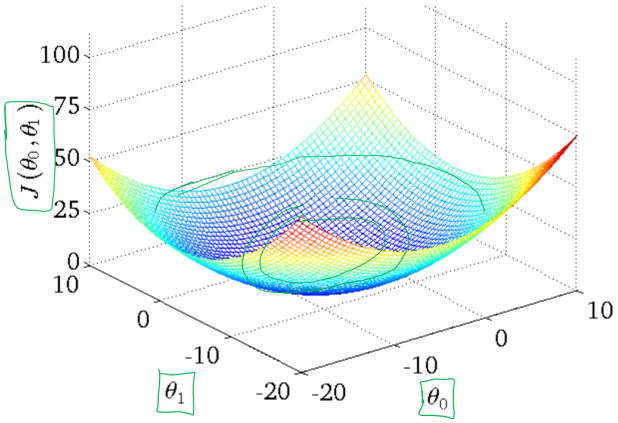


Figure 1.3: 3D surface plot of the cost function $J(\theta_0, \theta_1)$, showing the bowl-shaped function with the minimum at the center.

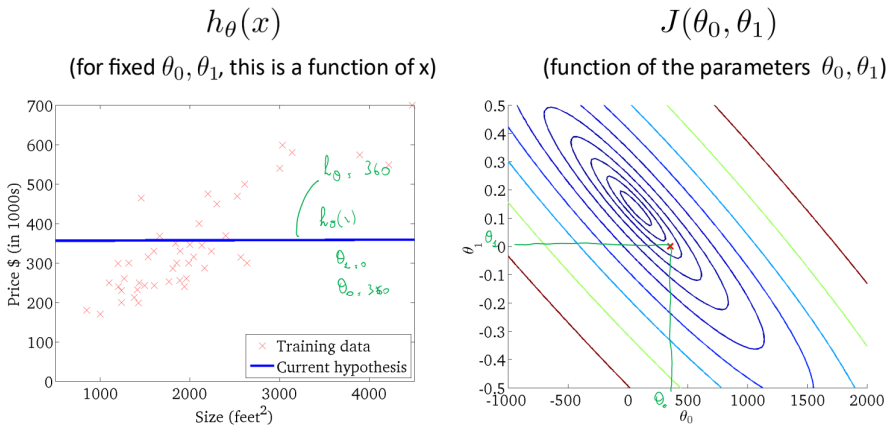


Figure 1.4: Contour plot of the cost function $J(\theta_0, \theta_1)$, showing the convergence path of gradient descent towards the minimum.

Note: Gradient descent might lead to a local minimum instead of the global minimum. Therefore, initialization is very important, and it is often recommended to set a seed for reproducibility.

1.3 Gradient Descent Algorithm

Gradient descent is an optimization algorithm used to minimize the cost function by iteratively updating the parameters θ in the direction of the steepest descent.

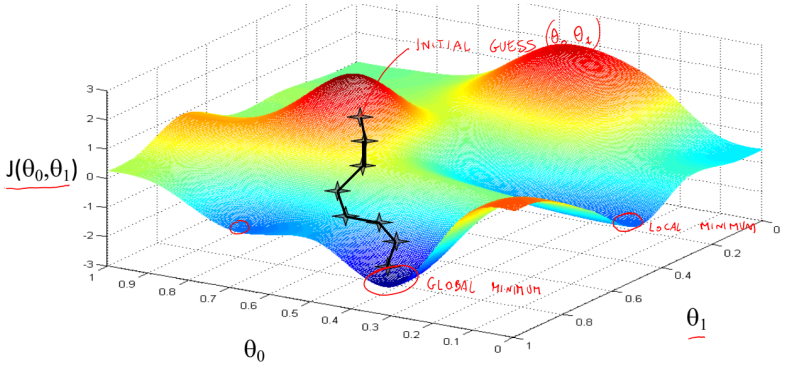


Figure 1.5: GD reaching Global minimum

1.3.1 Gradient Descent Update Rule

The goal of gradient descent is to minimize the cost function $J(\theta_0, \theta_1)$ by iteratively updating the parameters θ_0 and θ_1. The algorithm repeats until convergence:

$$\text{repeat until convergence } \left\{ \theta_j := \theta_j - \alpha \cdot \frac{\partial}{\partial \theta_j} J(\theta) \right. \tag{1.7}$$

Gradient Calculation: For linear regression, the gradient of the cost function is:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

This represents the direction and magnitude of the change needed for θ_j.

Gradient Descent Algorithm

The partial derivatives for $j = 0$ and $j = 1$ are as follows:

$$\theta_0 := \theta_0 - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \tag{1.8}$$

$$\theta_1 := \theta_1 - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)} \tag{1.9}$$

1.3.2 Linear Regression Model

In linear regression, we model the relationship between the input x and output y using a hypothesis function that is parameterized by θ_0 and θ_1:

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1.10}$$

The goal is to find values for θ_0 and θ_1 that minimize the cost function $J(\theta_0, \theta_1)$ with respect to each parameter. The general form of the partial derivative is:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{1.11}$$

where m is the number of training examples.

Visualize the process

Initially, we set the parameters θ_0 and θ_1 with arbitrary values. This initial hypothesis may not closely fit the data, resulting in a high cost $J(\theta_0, \theta_1)$.

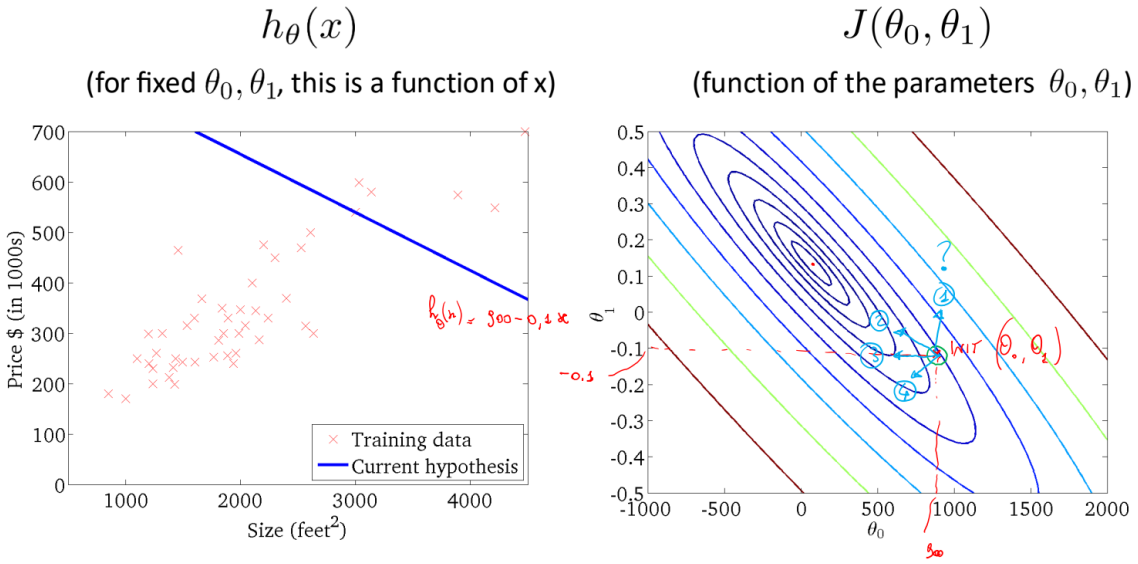


Figure 1.6: Initial Hypothesis and Cost Function Contours for Linear Regression

Next, we proceed to compute the gradients of the cost function with respect to each parameter, and iteratively adjust θ_0 and θ_1 using gradient descent. This process is repeated until we reach optimal values θ_0^* and θ_1^* that minimize $J(\theta_0, \theta_1)$.

The path of gradient descent, illustrates how the parameters are updated step by step, steadily approaching the minimum. At convergence, these optimized values θ_0^* and θ_1^* provide the best-fit line for the data, closely aligning with the observed data points and minimizing the prediction error.

1.3.3 Stochastic Gradient Descent (SGD)

For large training sets, calculating the gradient over all samples (like GD does) can be computationally expensive. Stochastic Gradient Descent (SGD) approximates the true gradient by using only one sample at a time.

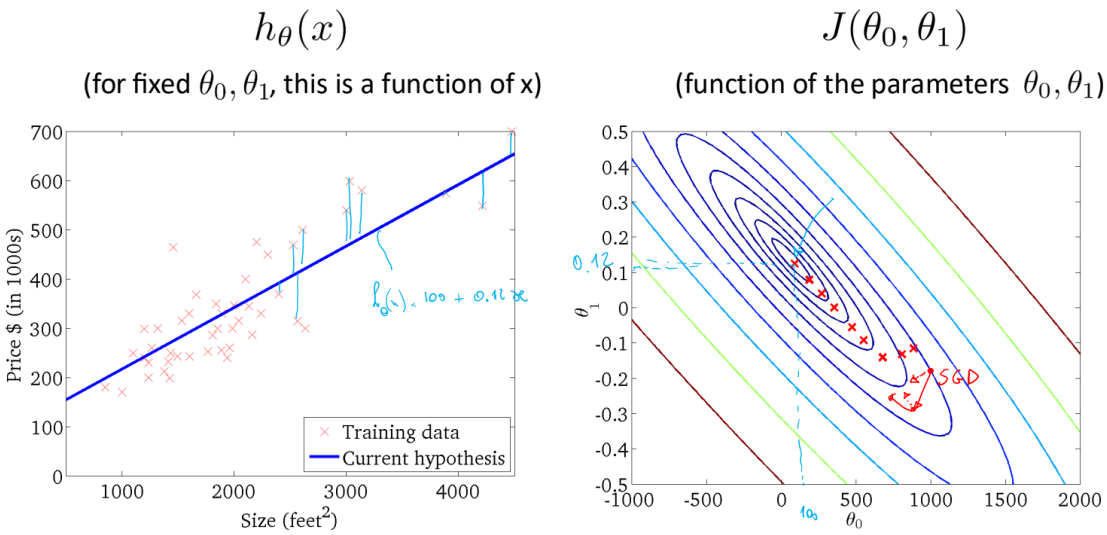


Figure 1.7: Path of Gradient Descent on Cost Function Contours

Update Rule:

$$\theta_j := \theta_j - \alpha \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$$

Where i is a randomly selected training example.

This method updates the parameters after evaluating each individual training example, allowing for more frequent parameter updates. This often results in faster initial progress toward minimizing the cost function, but can lead to a noisier path due to the variance introduced by each individual sample. This allow converge more quickly to a region near the optimal solution.

1.3.4 Mini-batch Gradient Descent

Mini-batch Gradient Descent is a compromise between Stochastic Gradient Descent (SGD) and Batch Gradient Descent. It splits the dataset into smaller "mini-batches" and computes the gradient on each mini-batch, rather than on the entire dataset or a single sample.

Update Rule:

$$\theta_j := \theta_j - \alpha \frac{1}{k} \sum_{i=1}^{k} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$$

Where k is the size of the mini-batch.

Advantages:

- This approach generally achieves smoother convergence than pure SGD, as it reduces the noise associated with single-sample updates.
- Mini-batch GD is typically faster and more efficient, as it leverages vectorized operations in modern libraries, which can process multiple samples at once.

Disadvantages:

- Requires tuning the mini-batch size.

The typical process includes:

1. Shuffling the data to ensure each epoch uses a different order of examples,
2. Dividing the data into mini-batches for each iteration,
3. Iterating through the entire dataset (one epoch) using these mini-batches,
4. Repeating the process until convergence.

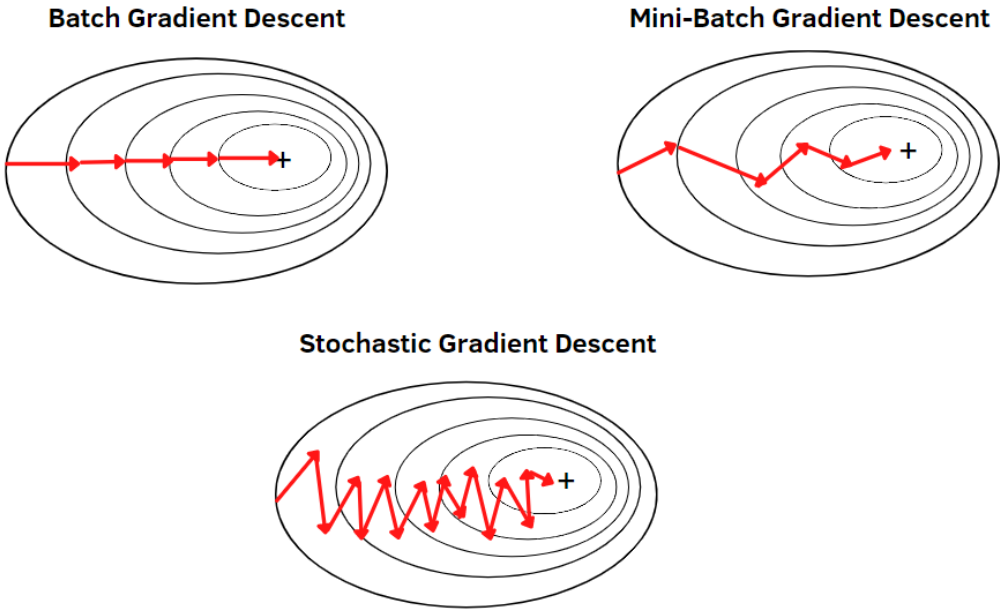
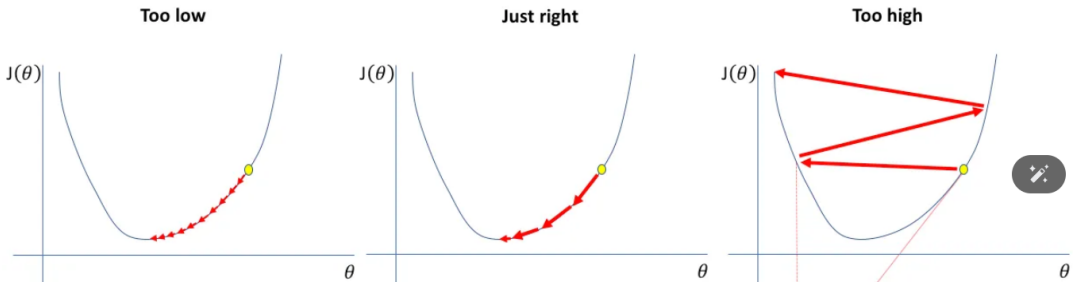


Figure 1.8: Comparison of Batch Gradient Descent, Mini-batch Gradient Descent, and Stochastic Gradient Descent

Learning Rate (α)

The convergence of gradient descent depends on the choice of the learning rate α, which control the step size:

- If α is too small, gradient descent can be very slow.
- If α is too large, gradient descent may overshoot the minimum, fail to converge, or even diverge.



1.3.5 Normal Equation

The normal equation provides an analytical solution to linear regression without requiring iteration. The parameters θ can be computed as:

$$\theta = (X^T X)^{-1} X^T y \tag{1.12}$$

where:

- X is the matrix of input features with dimensions $m \times n$,
- y is the vector of target values with dimensions $m \times 1$,
- $X^T X$ is an $n \times n$ matrix that must be invertible for the normal equation to be valid. This doesn't happen when we have:
  1. **Redundant Features:** Features are linearly dependent (e.g., size in feet^2 and size in meters^2).
  2. **Too Many Features:** More features than training examples ($n > m$). **Solution:** Regularization (e.g., Ridge Regression)

This method avoids the need to choose a learning rate, but the he computational cost of calculating $(X^T X)^{-1}$ is $O(n^3)$, which can be prohibitive for very large n.

1.2 Deriving the Normal Equation

The cost function for linear regression:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

can be written in matrix form:

$$J(\theta) = \frac{1}{2m} \left\| X\theta - y \right\|^2$$

To minimize $J(\theta)$, compute the gradient w.r.t. θ and set it to zero:

$$\nabla_\theta J(\theta) = \frac{1}{m} X^T (X\theta - y) = 0$$

Rearranging gives the **Normal Equation**

About Regularization

Regularization adds a penalty term to the cost function to prevent overfitting and ensure invertibility:

$$J(\theta) = \frac{1}{2m} \left\| X\theta - y \right\|^2 + \frac{\lambda}{2m} \|\theta\|^2$$

The Normal Equation with regularization becomes:

$$\theta = (X^T X + \lambda I)^{-1} X^T y$$

where $\lambda > 0$ is the regularization parameter, and I is the identity matrix.

4. Example: Applying the Normal Equation

Dataset:

- Input features: Size (in square feet), Number of Bedrooms.
- Target: Price (in $1000s).

$$X = \begin{bmatrix} 1 & 2104 & 5 \\ 1 & 1416 & 3 \\ 1 & 1534 & 3 \\ 1 & 852 & 2 \end{bmatrix}, \quad y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

Step 1: Compute $X^T X$ and $X^T y$:

$$X^T X = \begin{bmatrix} 4 & 5906 & 13 \\ 5906 & 11948228 & 29812 \\ 13 & 29812 & 87 \end{bmatrix}, \quad X^T y = \begin{bmatrix} 1185 \\ 2527810 \\ 6475 \end{bmatrix}$$

Step 2: Compute θ:

$$\theta = (X^T X)^{-1} X^T y$$

1.3.6 Probabilistic Interpretation of Least Squares

The least squares method can be understood from a probabilistic perspective by assuming a model with additive Gaussian noise.

- **Model Assumption**: Assume that each observation $y^{(i)}$ can be modeled as:

  $$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

  where $\epsilon^{(i)}$ represents the error term, capturing unmodeled effects and random noise in the observations.

- **Distribution of the Error Term**: We assume that $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$, meaning each error term is independently and identically distributed (i.i.d.) as a Gaussian with mean 0 and variance σ^2.

- **Conditional Probability of $y^{(i)}$ Given $x^{(i)}$**: Based on the model, the probability distribution of $y^{(i)}$ given $x^{(i)}$ and θ is:

  $$p(y^{(i)}|x^{(i)};\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$

  This expression shows that each $y^{(i)}$ is normally distributed around $\theta^T x^{(i)}$ with variance σ^2.

- **Likelihood of the Parameters θ**: The likelihood function $L(\theta)$ for all observations $\{y^{(i)}\}_{i=1}^m$ is the product of the individual conditional probabilities:

  $$L(\theta) = P(\mathbf{y}|X;\theta) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta)$$

  Expanding this, we get:

  $$L(\theta) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$

- **Log-Likelihood Function**: To simplify calculations, we take the natural logarithm of the likelihood function, obtaining the log-likelihood:

$$\ell(\theta) = \log L(\theta) = -\frac{m}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{m}(y^{(i)} - \theta^T x^{(i)})^2$$

- **Maximizing the Likelihood (MLE)**: To find the best-fitting parameters θ, we maximize the log-likelihood function. This is equivalent to minimizing the sum of squared errors (SSE):

$$\min_{\theta}\sum_{i=1}^{m}(y^{(i)} - \theta^T x^{(i)})^2$$

  This leads to the least squares estimate of θ, showing that minimizing SSE aligns with maximizing the likelihood under the Gaussian noise assumption.

This yields the optimal parameters θ.

1.3.7 Relationship between variables?

Correlation

When two variables exhibit a pattern such that they track each other's changes closely, it suggests a strong correlation between them. In the context of regression, this often leads to a model that fits the data points better.

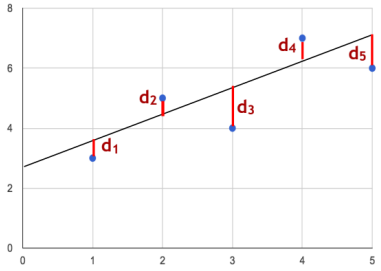
Method of least squares:

A primary measure for this relationship is the **Sum of Squared Errors (SSE)**, also known as the sum of squared residuals. In linear regression, we aim to find the line that minimizes the SSE, representing the best fit line for the data points.

Calculating Linear Regression using the Method of Least Squares:

- For a given data point and a line, the error is the vertical distance d from the point to the line, and the squared error is d^2.
- For a set of data points, the **Sum of Squared Errors (SSE)** is the sum of the squared errors for all points:

$$\text{SSE} = d_1^2 + d_2^2 + d_3^2 + \cdots + d_n^2$$



Pearson's Product-Moment Correlation (PPMC)

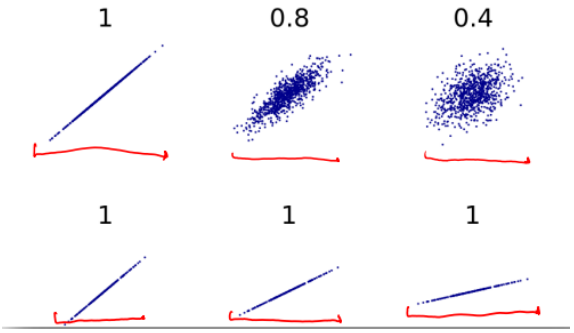
Often called the *Pearson coefficient* or *correlation coefficient*, this metric quantifies the linear correlation between two variables.

$$r = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}}$$

The coefficient r ranges from -1 to +1:

- $r = +1$: Maximum positive correlation, where the variables move in the same direction.
- $r = 0$: No correlation, indicating no linear relationship.
- $r = -1$: Maximum negative correlation, where the variables move in opposite directions.

Note: Swapping the x and y axes does not change the Pearson correlation coefficient, as it is symmetric.



Coefficient of Determination R^2

- R^2, or the *coefficient of determination*, measures the goodness of fit of a regression model, indicating how well the model explains the variability in the target variable.
- For simple linear regression, R^2 is calculated as the square of the Pearson correlation coefficient ($R^2 = r^2$).
- R^2 typically ranges from 0 to 1, with values closer to 1 indicating a better fit. An R^2 value of 1 signifies a perfect fit to the data points.
- This metric can be applied to any line or curve fit, allowing for the evaluation of models beyond linear regression.

1.3.8 Polynomial Regression

Polynomial regression extends linear regression by including polynomial terms of varying degrees (e.g., quadratic, cubic), allowing for more flexible models that can capture nonlinear relationships between variables.

The general form of the polynomial hypothesis function is:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \cdots + \theta_d x^d$$

where d is the degree of the polynomial.

Example of Polynomial Terms:
Linear term $\theta_1 x$, Quadratic term $\theta_2 x^2$, Cubic term $\theta_3 x^3$.

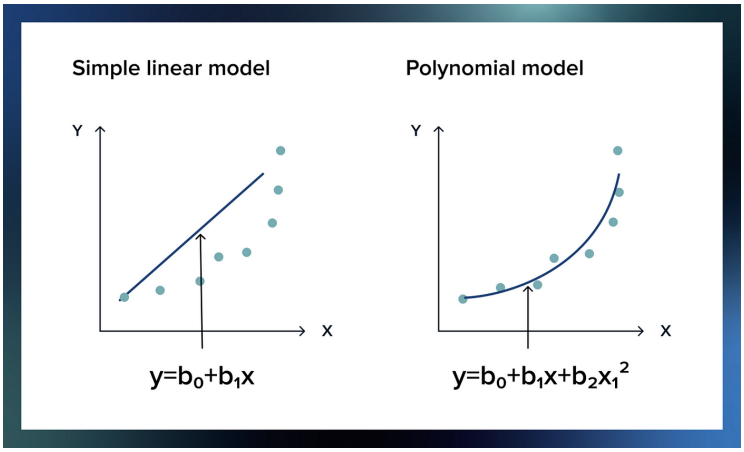


Figure 1.9: Polynomial Regression allows for a better fit to non-linear data points.

Overfitting and Underfitting:

- **Overfitting**: A high-degree polynomial can fit the training data very closely, capturing noise and fluctuations, which may lead to poor generalization on new data.
- **Underfitting**: A low-degree polynomial may not capture the underlying trends in the data, leading to a model that is too simplistic.

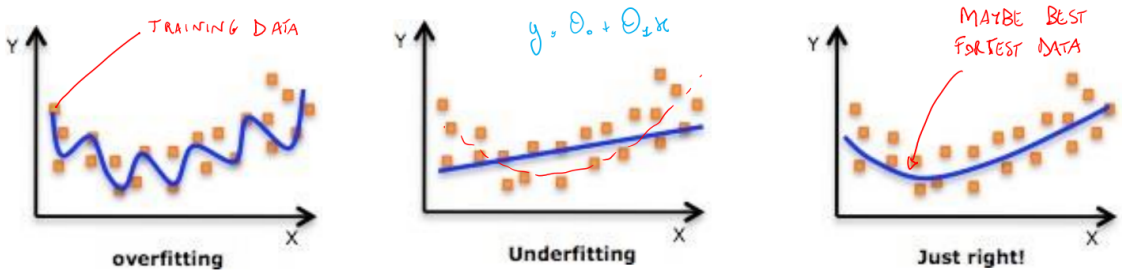


Figure 1.10: Choosing appropriate features is crucial for achieving a good balance between underfitting and overfitting.

Regularization in Polynomial Regression

1. **L2 Regularization (Ridge Regression):** Adds a penalty proportional to the square of the coefficients:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2$$

   Where λ controls the strength of regularization. This shrinks coefficients for higher-degree terms, reducing overfitting.

2. **L1 Regularization (Lasso Regression):** Adds a penalty proportional to the absolute values of the coefficients:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}|\theta_j|$$

   This encourages sparsity in the coefficients, effectively selecting features.

For more info everything is written perfectly here

1.3.9 Locally Weighted Regression (LWR)

Locally Weighted Regression (LWR) is a *non-parametric* regression technique. Unlike traditional regression models, LWR does not assume a fixed set of parameters that describe the entire dataset. Instead, it builds a local model around each query point, so nearby points influence the prediction more than distant points.
Process of Locally Weighted Regression:

1. **Build a local model** for each query point x_q. Instead of fitting one global model for all data points, LWR creates a diff model for each x_q.

2. **Assign higher weights** to data points closer to x_q, making the model more sensitive to local data around the query point.

3. **Repeat for all query points**: Each query point has its own local regression, producing as many models as there are query points.

Gaussian Kernel for Weighting:
A common weighting function in LWR is the *Gaussian kernel*, which defines the weight $w^{(i)}$ of each data point based on its distance d from the query point:

$$w^{(i)} = \exp\left(-\frac{d^2}{2\tau^2}\right)$$

where:

- d is $\|x^{(i)} - x\|$: Euclidean distance between $x^{(i)}$ and the query point x.
- τ: Bandwidth parameter controls how fast the weight drops with distance. Points closer to x_q have higher weights, while distant points contribute less to the local model.

Mathematics of LWR:

1. For a given query point x_q, the cost function $J(\theta)$ is weighted to emphasize nearby points:
$$J(\theta) = \sum_{i=1}^{m} w^{(i)} \left(y^{(i)} - \theta^T x^{(i)}\right)^2$$
where $w^{(i)}$ is calculated based on the distance between $x^{(i)}$ and x_q using the Gaussian kernel.

2. To compute the parameters θ that minimize the weighted cost function:
   (a) Rewrite the cost function in matrix form:
   $$J(\theta; x) = (X\theta - y)^T W (X\theta - y)$$
   (b) To minimize $J(\theta)$ for each query point, solve:
   $$\theta = (X^T W X)^{-1} X^T W Y$$
   where W is a diagonal matrix containing the weights $w^{(i)}(x)$

3. Compute θ specific to the query point x, yielding the prediction:
$$h_\theta(x) = \theta^T x$$

Considerations:

- **Advantages:**
  1. **Flexibility:** Adapts to non-linear relationships by fitting different local models.
  2. **No Assumption of a Global Model:** Eliminates bias introduced by a single, global hypothesis function.
- **Challenges:**
  1. **Computational Complexity:** Requires computing weights and solving the Normal Equation for each query point. Complexity scales with the dataset size $O(m^2)$, making it unsuitable for large datasets.
  2. **Storage Requirement:** Must retain the entire dataset in memory for predictions.

Chapter 2

Classification

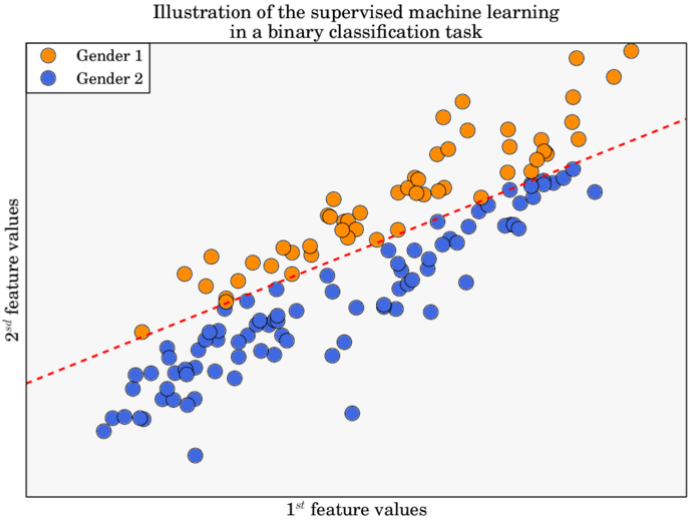
2.1 Binary Classification

2.1.1 Introduction

Binary classification is a supervised learning task where the objective is to assign instances to one of two classes. We denote:

- $y \in \{0, 1\}$: Target variable, where $y = 1$ represents the positive outcome and $y = 0$ the negative outcome (e.g., spam vs. not spam).
- Let $x \in \mathbb{R}^d$ represent the feature vector. The model $f_\theta(x)$ maps x to a predicted output $\hat{y}$.
- $f_\theta(x)$ represents the model's output, which belongs to the probability simplex Δ_c, indicating a probabilistic interpretation.
- $f_\theta(x) \in [0, 1]$: This output can be interpreted as the probability of $y = 1$.
- To classify, we use a threshold:

$$f_\theta(x) \geq 0.5 \Rightarrow \hat{y} = 1, \quad f_\theta(x) < 0.5 \Rightarrow \hat{y} = 0$$



Probability Model

For a given x, the probability that $y = 1$ is modeled as:

$$P(y = 1|x; \theta) = f_\theta(x)$$

where $f_\theta(x)$ could be a logistic function, defined as:

$$f_\theta(x) = \sigma(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

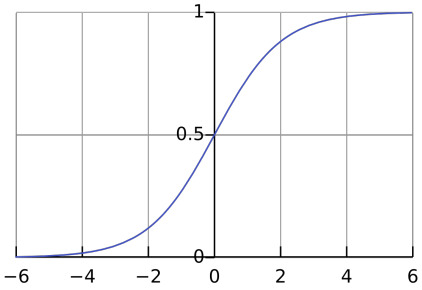
Then, $P(y = 0|x; \theta) = 1 - f_\theta(x)$.

2.1.2 Linear Decision Boundaries

For linear classifiers, the decision boundary is defined by a hyperplane:

$$\theta^T x = 0$$

For binary classification, the sigmoid function $\sigma(\theta^T x)$ transforms the decision boundary into probabilities. If $\sigma(\theta^T x) \geq 0.5$, we classify as $y = 1$, we have 0 otherwise.

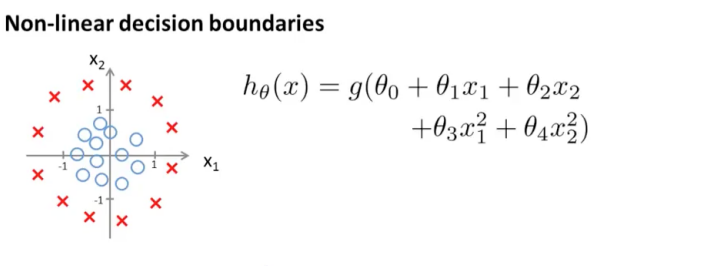


2.1.3 Non-Linear Decision Boundaries

Non-linear decision boundaries can be formed by adding higher-order terms or non-linear transformations of x, for example:

$$f_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2$$

This allows for more complex decision surfaces, such as circular or irregular shapes.



2.1.4 Bean Decision Boundary

In some cases, the data may require an even more complex, irregular decision boundary that resembles a "bean" shape or other non-standard contour. To achieve this, we can introduce higher-order polynomial terms or custom transformations, such as:

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2 + \theta_6 x_1^3 + \theta_7 x_2^3)$$

These higher-order terms allow the decision boundary to bend and form irregular, bean-like shapes that adapt to the data structure.

2.1.5 Likelihood and Loss Function

The likelihood function is:

$$P(Y|X;\theta) = \prod_{i=1}^{n} f_\theta(x_i)^{y_i}(1 - f_\theta(x_i))^{(1-y_i)}$$

The log-likelihood is then:

$$\log L(\theta) = \frac{1}{n}\sum_{i=1}^{n} (y_i \log f_\theta(x_i) + (1 - y_i)\log(1 - f_\theta(x_i)))$$

In practice, we minimize the Negative Log Likelihood (NLL):

$$\text{NLL}(\theta) = -\log L(\theta)$$

This ensures that as the probability for the correct class increases, the loss decreases.

The gradient of NLL with respect to θ is:

$$\frac{\partial \text{NLL}(\theta)}{\partial \theta} = -\frac{1}{n}\sum_{i=1}^{n}(y_i - \sigma(\theta^T x_i))x_i$$

Using Gradient Descent, we update θ as:

$$\theta \leftarrow \theta + \alpha\frac{1}{n}\sum_{i=1}^{n}(y_i - \sigma(\theta^T x_i))x_i$$

where α is the learning rate.

2.1.6 Newton's Method and Interpretation with Velocity and Acceleration

For faster convergence, Newton's Method can be applied:

$$\theta \leftarrow \theta - H^{-1}\nabla\text{NLL}(\theta)$$

where H is the Hessian matrix:

$$H = \sigma(\theta^T x_i)(1 - \sigma(\theta^T x_i))x_i x_i^T$$

Since computing H^{-1} is costly, quasi-Newton methods like BFGS can approximate the Hessian effectively.

In the context of Newton's method and optimization, we can interpret updates with concepts from physics:

- **Velocity** (θ'): This represents the rate of change of θ, indicating how quickly the parameter values are changing and in which direction.
- **Acceleration** (θ''): is the rate of change of velocity, that help to adjust to the step size in optimization adaptively.

2.2 Multi-class Classification

In multiclass classification, we deal with problems where the target variable can belong to one of C possible classes, with $C > 2$. Examples include classifying images of animals (e.g., "cat," "dog," "giraffe").

2.2.1 Softmax Function

The Softmax function is commonly used in multiclass classification to map output vectors to a probability simplex (mathematical space where each point represents a probability distribution between a finite number of mutually exclusive category). The Softmax function for a class i is defined as:

$$\text{softmax}(a_i) = \frac{\exp(a_i)}{\sum_j \exp(a_j)}$$

where a_i is the logit or raw output score for class i.

- $\exp(a_i)$: Ensures that higher a_i results in a higher probability for class i.
- Summation constraint $\sum_j \exp(a_j)$: Ensures all probabilities sum to 1, maintaining a valid probability distribution.

The output probability vector $\mathbf{f}(\mathbf{x})$ represents the likelihood of each class, encoding the distribution over all classes.

2.2.2 Probabilistic Interpretation and Cross-Entropy Loss

Given a one-hot encoded target vector y, where $y = [0, 1, 0]$ might represent class 2 in a 3-class problem, we can interpret the prob distribution over classes using $\mathbf{f}(\mathbf{x})$:

$$P(Y = i|\mathbf{x}) = f(\mathbf{x})_i$$

The likelihood function for a single data point i is then:

$$P(Y^{(i)}|\mathbf{x}^{(i)}; \theta) = \prod_j f(\mathbf{x}^{(i)})_j^{y_j^{(i)}}$$

where only the probability corresponding to the true class $y_j^{(i)} = 1$ contributes to the likelihood.

Cross-Entropy Loss: The negative log-likelihood of the Softmax output, also called the Cross-Entropy (CE) Loss, is given by:

$$\text{CE}(\theta) = -\sum_{i=1}^{n} \sum_{j=1}^{C} y_j^{(i)} \log f(\mathbf{x}^{(i)})_j$$

For a correctly classified example, the log probability of the predicted class is maximized, while the log probabilities for incorrect classes do not contribute, minimizing the loss.

2.2.3 Entropy, Cross-Entropy, and KL-Divergence

- **Entropy** $H(p)$: Measures the uncertainty of a probability distribution p over classes.
$$H(p) = -\sum_i p_i \log p_i$$

- **Cross-Entropy CE**(p, q): Measures the discrepancy when encoding p with a different distribution q.
$$\text{CE}(p, q) = -\sum_i p_i \log q_i$$

- **Kullback-Leibler Divergence (KL-Divergence)**: Represents the additional cost of using q to approximate p.
$$KL(p||q) = \sum_i p_i \log \frac{p_i}{q_i}$$

Relationship: The Cross-Entropy can be decomposed as:

$$\text{CE}(p, q) = H(p) + KL(p||q)$$

indicating that Cross-Entropy accounts for the inherent uncertainty in p and the cost of approximation using q.

2.2.4 Expected Calibration Error (ECE)

ECE is a measure of model calibration, which assesses whether the confidence of predictions matches the actual accuracy.

1. **Divide** the interval $[0, 1]$ into m bins.
2. **Calculate** the number of samples B_m whose confidence falls into each bin.
3. For each bin:
   - p_m: Average confidence for bin m.
   - a_m: Accuracy for bin m.
4. **Expected Calibration Error**:
$$\text{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{n} |a_m - p_m|$$

2.2.5 Bias-Variance Trade-Off

In machine learning, the **bias-variance trade-off** is a key concept to understand the sources of error in model predictions. This trade-off helps us balance underfitting and overfitting, thus improving model generalization on unseen data.

Bias

Bias refers to the error introduced by approximating a real-world problem, which may be complex, by a simpler model. High bias implies a model is too simple and fails to capture the underlying patterns in the data, leading to underfitting. In this case, the model predictions are far from the actual data points, showing a consistent deviation from the true values.

Variance

Variance refers to the model's sensitivity to small fluctuations in the training dataset. A high-variance model pays too much attention to training data, capturing noise as if it were signal, which leads to overfitting. In this case, the model performs well on training data but poorly on unseen data, as it fails to generalize.

Trade-Off

The error in a model's prediction can be decomposed into:

$$\mathbb{E}[(y - f(x))^2] = \text{Bias}(f(x))^2 + \text{Variance}(f(x)) + \sigma^2$$

where σ^2 represents irreducible error due to noise in the data.
The goal is to minimize both bias and variance, but typically, reducing one increases the other. This trade-off is illustrated in the **bias-variance curve**, where we see that as model complexity increases, bias decreases, but variance increases.

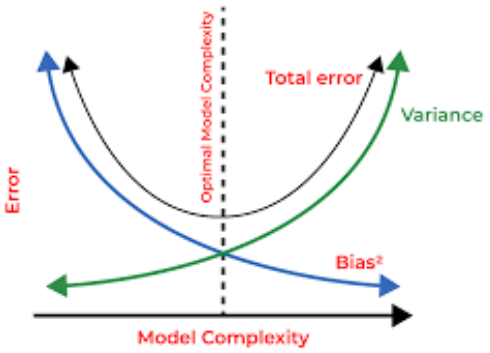
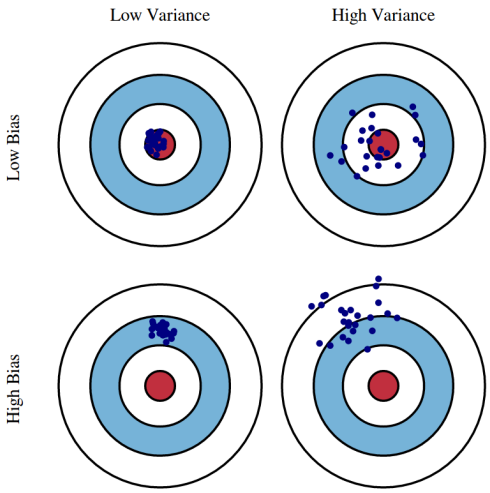


Figure 2.1: Trade off between Bias and Variance

- **Underfitting (High Bias)**: When the model is too simple (e.g., linear fit on non-linear data), fails to capture the true trend in the data.
- **Overfitting (High Variance)**: When the model is too complex (e.g., polynomial with too high a degree), it fits the training data well but fails to generalize to new data.



Example on Polynomial Regression

Consider fitting a polynomial $\hat{f}(\mathbf{x})$ of degree d to a dataset:

- **Low Degree Polynomial (High Bias)**: The model is too simple to capture the curvature of the true function $f(\mathbf{x})$. Predictions are consistently far from the true function, resulting in high bias.
- **High Degree Polynomial (High Variance)**: The model perfectly fits the training data, including noise. Predictions vary significantly with slight changes in the training data, resulting in high variance.
- **Optimal Degree**: A degree that captures the true structure of $f(\mathbf{x})$ without overfitting noise.

2.2.6 Hyperplanes in Machine Learning

In N-dimensional space, a **hyperplane** is a flat subspace with $N-1$ dimensions. Hyperplanes are commonly used in classification tasks, particularly in linear classifiers like Support Vector Machines (SVMs).

Hyperplane in Different Dimensions

- In a 2D space, a hyperplane is a 1D line.
- In a 3D space, a hyperplane is a 2D plane.
- In N-dimensional space, a hyperplane is an $(N-1)$-dimensional subspace.

Properties and Use in Classification

- **Separating Classes**: In classification, a hyperplane can act as a decision boundary, separating data points of different classes. A linear classifier finds a hyperplane that best separates the classes if they are linearly separable.
- **Margin**: The distance from the closest data points to the hyperplane is called the margin. Support Vector Machines (SVM) aim to maximize this margin, improving the model's robustness and generalization.

2.2.7 Probability Distributions in Multi-Class Classification

Let $f(x)$ encode the probability distribution over all the classes. $f(x)_i$ represents the probability for a single class i.

$$P(Y \mid f(x)) = \prod_{i=1}^{C} [f(x)_i]^{y_i}$$

Since Y is a one-hot vector, only $f(x)_y$ (the probability of the correct class) contributes to the product. Thus, we can simplify to:

$$P(Y \mid f(x)) = f(x)_y$$

Likelihood and Log-Likelihood

The likelihood function L is defined as:

$$L = \prod_{i=1}^{n} P(Y^i \mid f(x^i))$$

The log-likelihood is then given by:

$$\log L = \sum_{i=1}^{n} \log P(Y^i \mid f(x^i)) = \sum_{i=1}^{n} \log f(x^i)_{y_i}$$

Negative Log-Likelihood (NLL) and Cross-Entropy

The negative log-likelihood (NLL) can be defined as:

$$\text{NLL} = -\sum_{i=1}^{n} \log f(x^i)_{y_i} \quad = \text{Cross Entropy}$$

Expanding this over all classes C:

$$= -\sum_{i=1}^{n} \sum_{j=1}^{C} y_j^i \log f(x^i)_j$$

Softmax and Numerical Stability

The goal is to maximize the probability of the true class at the expense of other outputs. The softmax function can be unstable. To avoid numerical issues, we use:

$$-\log \left(\frac{\exp(p_i)}{\sum_j \exp(p_j)} \right) = -p_i + \log \sum_j \exp(p_j)$$

To ensure stability, we subtract the maximum c from each p_i:

$$\log \left(\sum_j \exp(p_j - c) \right) + c \quad \text{where } c = \max(p)$$

This adjustment ensures numerical instabilities do not occur.

Mean Squared Error Decomposition into Bias and Variance

$$\begin{aligned}\mathbb{E}[(y-\hat{f}(x))^2] &= \mathbb{E}\big[(g(x)+\epsilon-\hat{f}(x))^2\big] = \\ &= \mathbb{E}\big[(g(x))^2\big] + \mathbb{E}\big[\epsilon^2\big] + \mathbb{E}\big[\hat{f}(x)^2\big] - \\ &\quad - 2\mathbb{E}[g(x)\hat{f}(x)] - 2\mathbb{E}[\epsilon\hat{f}(x)] \quad - 2\mathbb{E}[g(x)\epsilon] = \\ &= g(x)^2 + \sigma^2 + \mathbb{E}[\hat{f}(x)^2] - \\ &\quad - 2g(x)\mathbb{E}[\hat{f}(x)] - 2\mathbb{E}[\epsilon\hat{f}(x)] \quad - 2g(x)\mathbb{E}[\epsilon].\end{aligned}$$

Assumptions:

1. $\mathbb{E}[\epsilon] = 0$: The noise has zero mean.
2. $\mathbb{E}[\epsilon^2] = \text{Var}(\epsilon) = \sigma^2$: The variance of the noise.
3. $\mathbb{E}[\hat{f}(x)] = g(x)$: The expectation of the predicted function matches the true function, meaning $g(x)$ is independent of the dataset of choice.

Substituting these assumptions:

$$\begin{aligned}\mathbb{E}[(y-\hat{f}(x))^2] &= g(x)^2 + \sigma^2 + \mathbb{E}[\hat{f}(x)^2] - 2g(x)\mathbb{E}[\hat{f}(x)] = \\ &= g(x)^2 + \sigma^2 + \mathbb{E}[\hat{f}(x)^2] - 2g(x)\mathbb{E}[\hat{f}(x)].\end{aligned}$$

Expanding further:

$\text{Bias}^2(\hat{f}(x)) = (g(x) - \mathbb{E}[\hat{f}(x)])^2$, (error between true and expected prediction)
$\text{Variance}(\hat{f}(x)) = \mathbb{E}[\hat{f}(x)^2] - \mathbb{E}[\hat{f}(x)]^2$, (variability across datasets).

Final decomposition:

$$\mathbb{E}[(y-\hat{f}(x))^2] = \text{Bias}^2(\hat{f}(x)) + \text{Variance}(\hat{f}(x)) + \sigma^2.$$

2.2.8 Regularization Overview

Regularization is a technique used to prevent overfitting by adding a penalty term to the objective function. This additional term discourages complex models by penalizing large parameter values, ensuring that the learned model generalizes well to unseen data.

Types of Regularization

2.2.9 1. Ridge Regression (L2 Regularization)

Ridge regression adds the squared magnitude of the model parameters as a penalty to the loss function. This penalizes large parameter values, reducing model complexity.

The loss function for Ridge regression is:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f_\theta(x_i))^2 + \alpha\sum_{j=1}^{P}\theta_j^2$$

$\alpha = 0$: No regularization (ordinary least squares).

2. Lasso Regression (L1 Regularization)

Lasso regression adds the absolute magnitude of the model parameters as a penalty. This tends to produce sparse solutions, where some coefficients are exactly zero.

The loss function for Lasso regression is:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f_\theta(x_i))^2 + \alpha\sum_{j=1}^{P}|\theta_j|$$

Lasso is useful for feature selection, as it shrinks some coefficients to zero.

3. Elastic Net Regularization

Elastic Net combines L1 (Lasso) and L2 (Ridge) penalties. It balances the strengths of both methods, penalizing both the magnitude and the sparsity of parameters.

The loss function for Elastic Net is:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f_\theta(x_i))^2 + \alpha_1\sum_{j=1}^{P}|\theta_j| + \alpha_2\sum_{j=1}^{P}\theta_j^2$$

Elastic Net is effective when there are highly correlated features.

Effect of Regularization Parameter (α)

The regularization parameter α controls the trade-off between minimizing the loss function and penalizing model complexity:

- Small α: Model is more complex, risk of overfitting.
- Large α: Model is simpler, risk of underfitting.

2.2.10 Probabilistic Interpretation of Regularization

For Ridge regression:

$$\mathcal{L}(\theta) = \text{Negative log-likelihood of data} + \frac{\lambda}{2}||\theta||_2^2$$

This corresponds to placing a Gaussian prior on the parameters θ:

$$P(\theta) \sim \mathcal{N}(0, \sigma^2 I)$$

For Lasso regression:

$$\mathcal{L}(\theta) = \text{Negative log-likelihood of data} + \lambda||\theta||_1$$

This corresponds to placing a Laplace prior on the parameters θ.

Chapter 3

Introduction to Backpropagation

3.0.1 What is Backpropagation?

Backpropagation, short for **backward propagation of errors**, is an algorithm used to train neural networks. It calculates the gradient of the loss function with respect to each weight in the network. These gradients are then used to update weights through optimization methods like gradient descent.

3.1 Math Framework for Backpropagation

Components of Gradient Computation

The backpropagation algorithm relies on:

1. **Nonlinear score function:**

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x), \tag{3.1}$$

   where $\max(0, W_1 x)$ applies the ReLU activation, which outputs 0 if the input is negative and the input itself if positive.

2. **Softmax loss function:**

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right), \tag{3.2}$$

   where s_{y_i} is the score for the true class y_i. This loss is suitable for multi-class classification problems.

3. **Regularization term:**

$$R(W) = \sum_k W_k^2, \tag{3.3}$$

   which reduces overfitting by penalizing large weights.

4. **Total Loss** The total loss is a combination of data loss and regularization:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda\left(R(W_1) + R(W_2)\right), \tag{3.4}$$

   where λ controls the trade-off between the two terms.

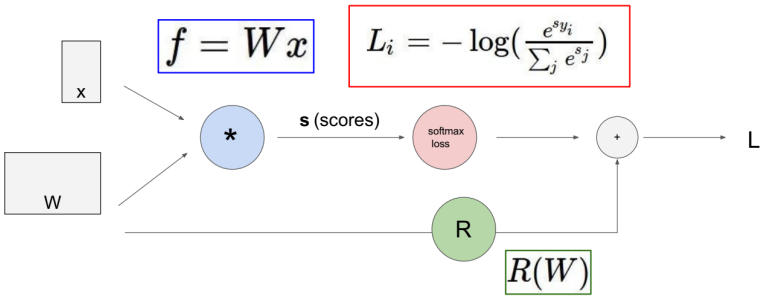


Figure 3.1: Computational Graphs + Backpropagation.

3.2 Backpropagation Algorithm

Forward Pass

The forward pass computes the network's output by evaluating operations layer by layer. Intermediate values are stored for the backward pass.

Backward Pass

The backward pass computes gradients by applying the chain rule:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial q} \cdot \frac{\partial q}{\partial x}. \tag{3.5}$$

3.2.1 Example: Simple Function

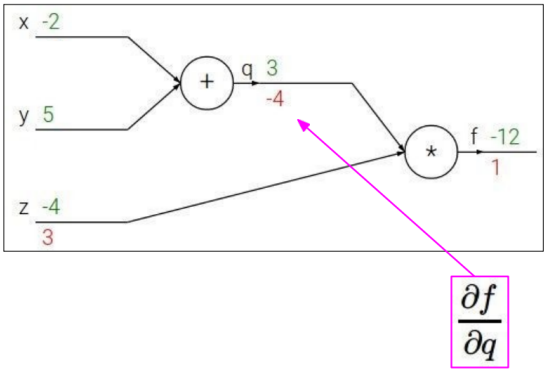
Function: $f(x, y, z) = (x + y)z$ **Steps:**

1. Compute $q = x + y$. So

$$\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = q \tag{3.6}$$

2. Compute $f = qz$. So

$$\frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q \tag{3.7}$$



3. Chain Rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y}$$

- **Upstream gradient:** $\frac{\partial f}{\partial q}$
- **Local gradient:** $\frac{\partial q}{\partial y}$

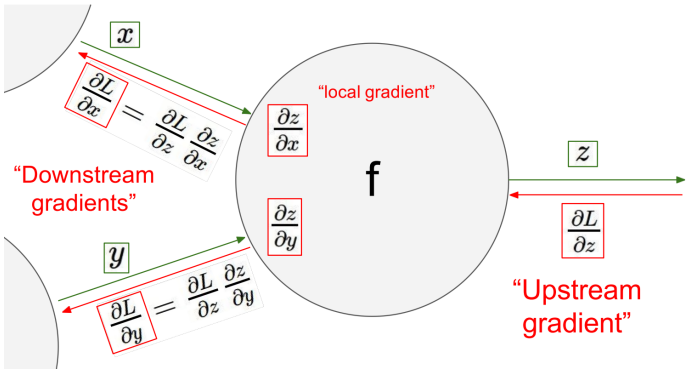


Figure 3.2: Illustration of the chain rule in backpropagation

The diagram shows a computational graph for a function f, where gradients propagate backward. The upstream gradient ($\frac{\partial L}{\partial z}$) flows from the output of the node. The local gradients ($\frac{\partial z}{\partial x}$, $\frac{\partial z}{\partial y}$) represent the partial derivatives of the node's output with respect to its inputs. The downstream gradients ($\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial y}$) are computed using the chain rule, enabling parameter updates in the optimization process.

3.2.2 Example 2: Sigmoid Function

Function: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$

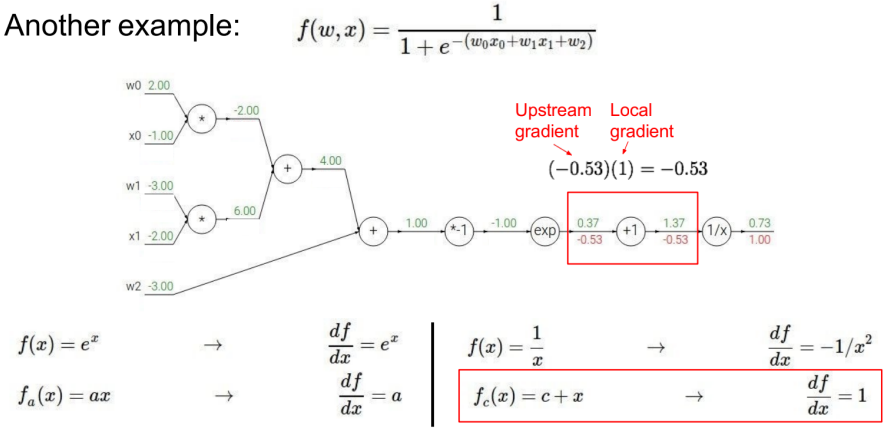


Figure 3.3: example of the forward and backward pass in a computational graph for the function $f(w,x)$.

Forward Pass

- Inputs x_0, x_1 and weights w_0, w_1, w_2 are passed through a series of operations (multiplication, addition, exponentiation, reciprocal). Intermediate values for each node are computed step-by-step.
- The output $f(w, x)$ is computed through a sequence of operations.

Backward Pass

The backward pass propagates gradients backward using the chain rule:

- Starting with the loss gradient $\frac{\partial L}{\partial f}$, the chain rule is applied at each node:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x}$$

- **Upstream Gradient**: The gradient received from the previous node (e.g., -0.53 at the exponentiation node).
- **Local Gradient**: The derivative of the operation at each node (e.g., $\frac{\partial z}{\partial x}$ for addition or multiplication).

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right) \cdot \frac{1}{1+e^{-x}} = \sigma(x) \cdot (1-\sigma(x))$$

Derivative Rules

The derivatives of basic operations, such as:

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x, \quad f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -\frac{1}{x^2},$$

are used to compute the gradients efficiently during the backward pass.

3.3 Neural Networks

Neural networks, also referred to as **multi-layer perceptrons (MLPs)**, consist of layers of neurons with learnable parameters (weights and biases).

3.3.1 Linear vs. Neural Networks

Linear score function:

$$f = W \cdot x$$

2-layer Neural Network:

$$f = W_2 \cdot \text{ReLU}(W_1 \cdot x)$$

Where:

- W_1, W_2**:** Weight matrices. **ReLU:** Activation function introducing non-linearity.

Deeper Networks

- Networks can extend to deeper architectures, such as:
  - **3-layer Neural Networks**: One additional hidden layer.
  - **Deep Networks**: Many layers for hierarchical feature extraction.
- Larger models can capture more complex patterns but require careful regularization.

Learning Templates

A neural network with a hidden layer learns **templates** for input patterns.

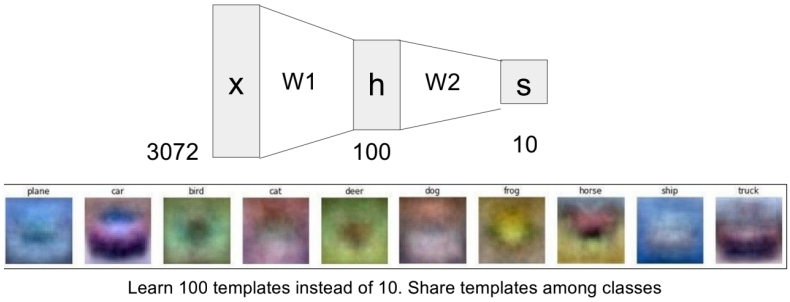


Figure 3.4: Instead of learning 10 class-specific templates, the network can learn 100 shared templates that apply across multiple classes.

3.3.2 Importance of Activation Functions

Activation functions introduce non-linearity, enabling networks to learn complex patterns. Without them, the network would behave like a linear classifier regardless of depth. Common activation functions include:

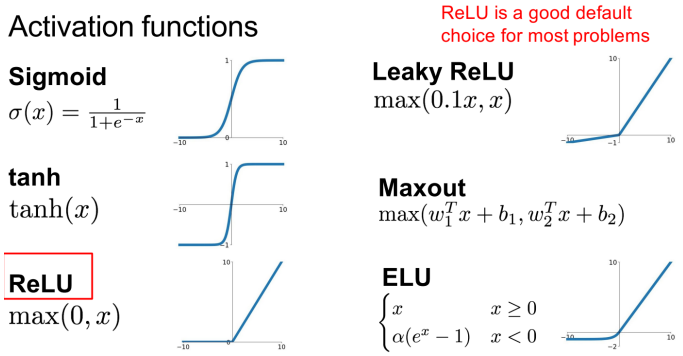


Figure 3.5: Various Activation Functions and Their Characteristics.

Universal Approximation Theorem

> Let $h(x)$ be a continuous function on a compact subset $S \subseteq \mathbb{R}^d$. For any $\epsilon > 0$, there exists a NN $f(x)$ with sufficiently many hidden units such that:
>
> $$|h(x) - f(x)| < \epsilon, \quad \forall x \in S$$

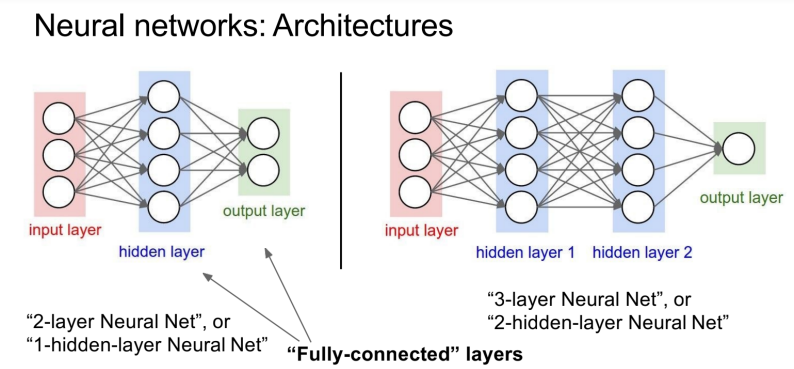


Figure 3.6: NN Architecture

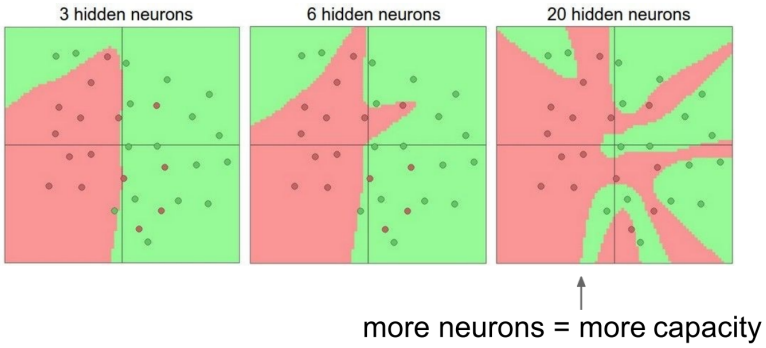


Figure 3.7: Setting Number of Layers and their Size

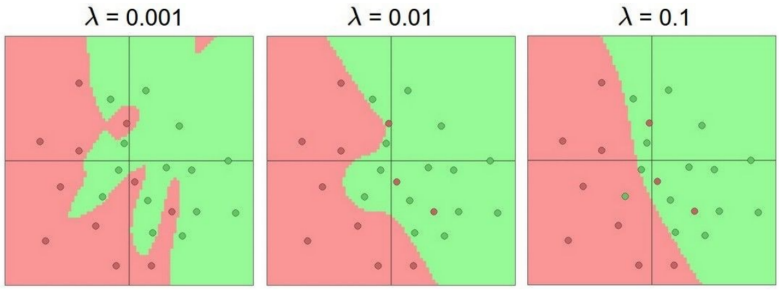
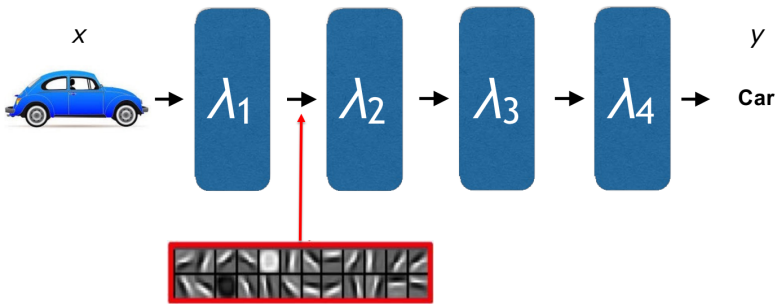


Figure 3.8: Importance of Regularization

3.3.3 Deep Learning Features

Deep networks automatically learn features from data. Feature extraction becomes parameterized:

$$\text{features} = g(x, \lambda), \quad \text{output} = f(\text{features}, \omega)$$



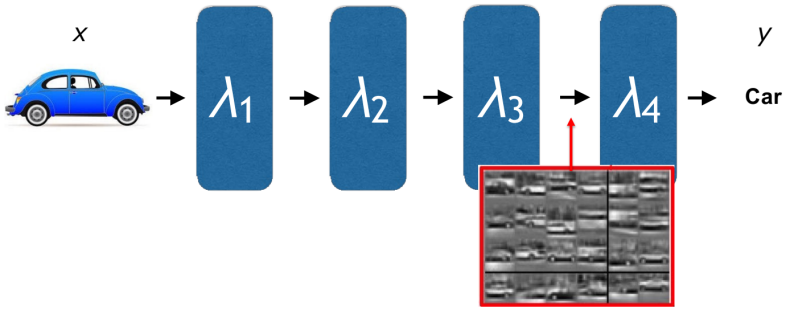


Figure 3.9: Each intermediate representations learn templates of growing complexity(details)

3.3.4 Deep Learning: End-to-End Systems

- All components are trainable, eliminating the boundary between feature extraction and classification.
- Non-linear transformations are applied across layers to map input data to the desired output.
- Deep architectures use **building blocks**, such as neurons, to compose complex systems, where each block has trainable parameters (λ_i).

Overview of Training Process

- During the **forward pass**, the network produces predictions for a given input.
- The **loss function** measures the error between predictions and true labels.
- In the **backward pass**, **errors are propagated** back through the network using gradients to update weights.

Challenges in Training

- Training is **highly dependent on initialization**.
- The optimization landscape is **non-convex**, with multiple local minima and saddle points.
- Achieving the global optimum is often impractical.

Chapter 4

Convolutional Neural Networks

CNN (ConvNets) are a class of neural networks that excel in processing data with grid-like topology, such as images. They achieve remarkable performance by learning spatial hierarchies of features directly from data.

4.1 Convolutional Layers

4.1.1 Preserving Spatial Structure

Unlike fully connected layers, convolutional layers process data in chunks to preserve spatial hierarchies:

- Input: A 3D volume (e.g., a $32 \times 32 \times 3$ image).
- Filter: A 3D kernel (e.g., $5 \times 5 \times 3$) slides across spatial locations.
- Output: An **activation map** representing responses to the filter.

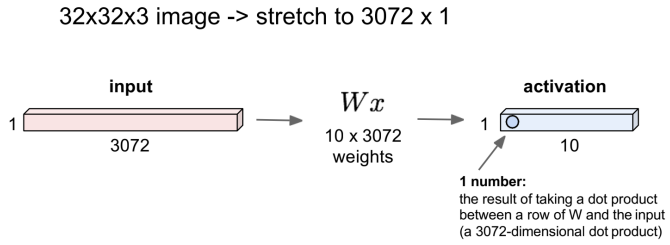


Figure 4.1: Convolution Layer Processing.

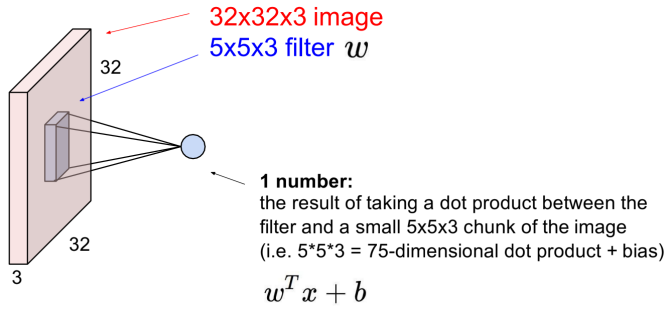


Figure 4.2: convolve (slide) over all spatial locations

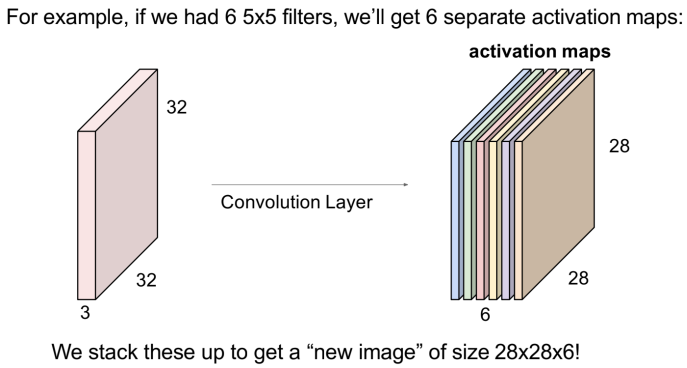


Figure 4.3: Repeat for how many filters u need

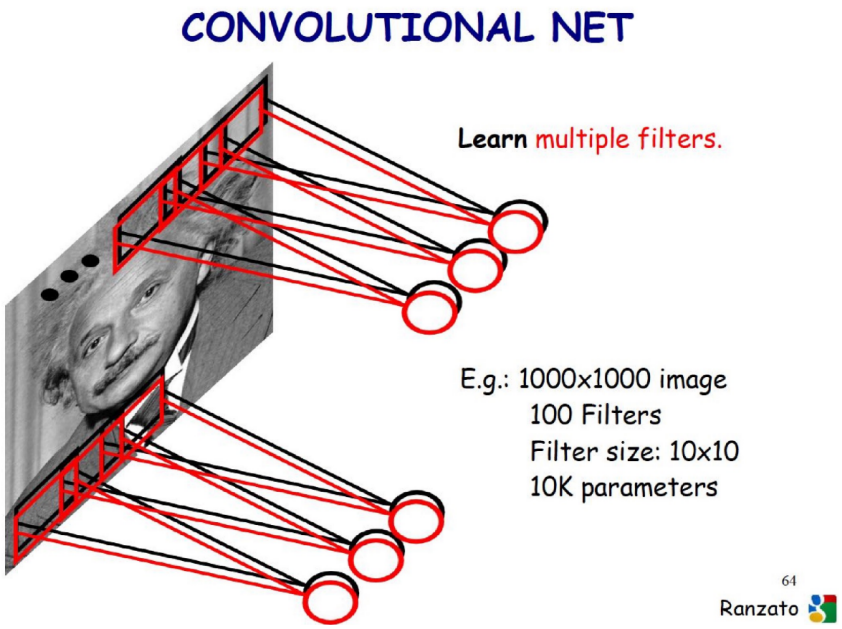


Figure 4.4: Convolution Layer Processing.

4.1.2 Hyperparameters in Convolutional Layers

- **Number of Filters (K)**: Determines depth of output volume.
- **Filter Size (F)**: Size of the kernel (e.g., 3×3 or 5×5).
- **Stride (S)**: Determines step size of the filter.

- **Zero Padding (P)**: Adds padding around the input to control spatial dimensions.

4.1.3 Example of Convolution

For a $32 \times 32 \times 3$ image convolved with a $5 \times 5 \times 3$ filter:

- Each filter produces a single number from a dot product.
- Output activation map dimensions depend on S and P.

4.2 Activation Maps and Stacking

- Multiple filters create multiple activation maps.
- These maps are stacked to form a new volume (e.g., $28 \times 28 \times 6$).

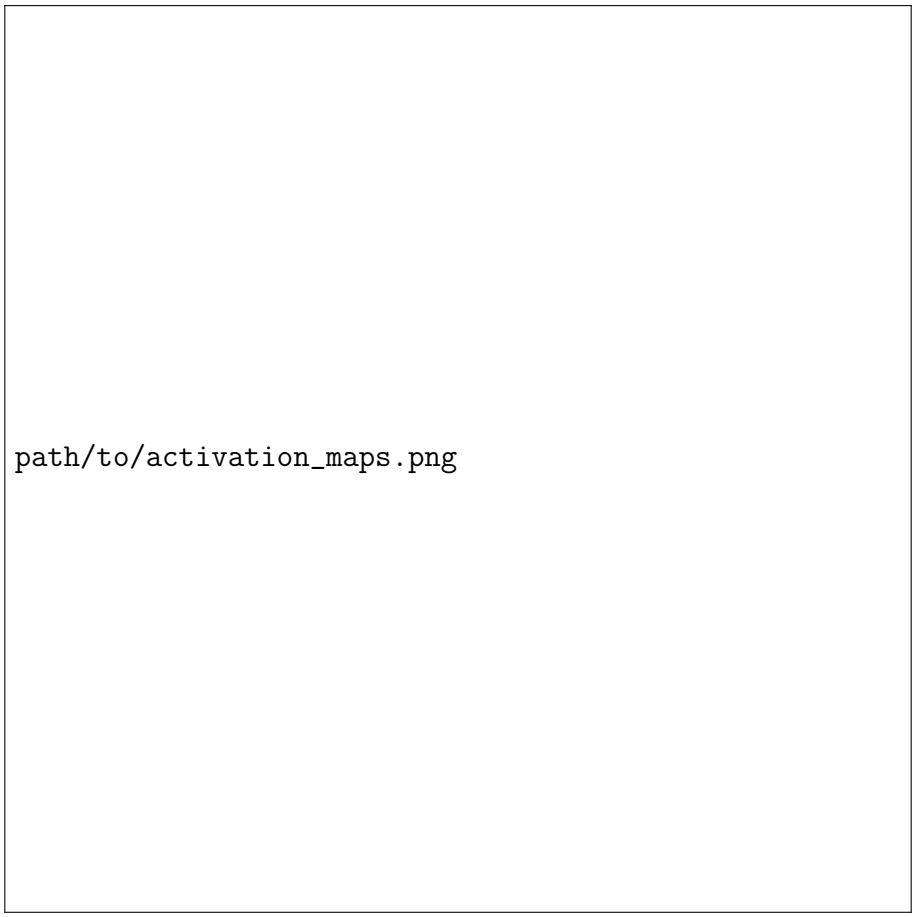


Figure 4.5: Stacking Activation Maps.

4.3 ConvNet Structure

4.3.1 Common Layers

- **Convolution (CONV)**: Extracts spatial features.

- **ReLU Activation**: Introduces non-linearity.
- **Pooling (POOL)**: Reduces spatial dimensions while retaining key features.
- **Fully Connected (FC)**: Connects neurons to all activations in the previous layer.

Figure 4.6: Typical ConvNet Pipeline: CONV, ReLU, POOL, FC.

4.3.2 Pooling Layers

- **Max Pooling**: Captures the maximum value within a window (e.g., 2×2).
- Reduces spatial dimensions and introduces **spatial invariance**.
- Example:

$$\begin{bmatrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} \xrightarrow{\text{Max Pooling (2x2, Stride=2)}} \begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}$$

4.4 Advances in ConvNet Architectures

4.4.1 Trends

- Smaller filters (e.g., 3×3) and deeper architectures.

- Replacing POOL/FC layers with additional CONV layers (e.g., ResNet, GoogLeNet).
- Historical architectures followed the pattern:

$$[(\text{CONV-RELU})^N - \text{POOL}]^M - (\text{FC-RELU})^K - \text{SOFTMAX}$$

  Where $N \leq 5$, M is large, and $K \in [0, 2]$.

4.4.2 Visualization of Features

- Features learned by ConvNets at different layers can be visualized.
- Earlier layers capture edges and textures, while deeper layers capture object parts.

Figure 4.7: Visualization of Features in ConvNet Layers.

4.5 Summary

- ConvNets combine CONV, POOL, and FC layers to process data hierarchically.
- The trend is moving towards deeper architectures with fewer POOL/FC layers.
- State-of-the-art models (e.g., ResNet) use skip connections to mitigate gradient vanishing problems.

4.6 Acknowledgments

Material sourced from:

- **Bernt Schiele, Mario Fritz, Fei-Fei Li, Justin Johnson, Serena Yeung, Rob Fergus**.
- **Zeiler and Fergus (2013)**: Feature visualization techniques.