AMATH 445: Scientific Machine Learning Notes

Pratham Chauhan

February 12, 2026

Contents

1 Lecture 1: Introduction to machine learning

1.1 What is machine learning?

Definition 1. An approach to making decisions or predictions based on data, rather than explicit instructions. The algorithm improves its performance on a task through experience (i.e., through data).
For a normal program, we have:

$$\text{Input data} + \text{Recipe} \rightarrow \text{traditional program} \rightarrow \text{Output}$$

For a machine learning program, we have:

$$\text{Input data} + \text{Output} \rightarrow \text{machine learning algorithm} \rightarrow \text{Recipe}$$

Example.

- Spam email detection
- Generating text/images
- Optimal game playing (e.g., chess, Go)

1.2 Factors driving the growth of machine learning

- Availability of large datasets:
  - imageNet-1k has over 1.3 million labeled images across 1000 categories
  - MNIST dataset has 70,000 labeled images of handwritten digits (10 classes (0-9))
- Cheap parallel processing/computeing resources (GPUs, TPUs)
- Algorithms and infrastructure maturity
  - CNN, RNN, transformers
  - open source libraries (TensorFlow, PyTorch, Scikit-learn)
  - proven success (ChatGPT)

1.3 What can we do with machine learning?

- Prediction: Forecasting unknown outcomes (e.g. temperature prediction, stock prices)

- Representation: Finding structure in data (e.g. clustering, dimensionality reduction)
- Decision making: Choosing optimal actions based on data (e.g. robotics, game playing)
- Generation: Creating new content (e.g. text generation, image synthesis)
- Hybrid tasks: Combining multiple objectives (e.g. autonomous driving, recommendation systems)

1.4 Advantages of employing machine learning

Remark. When do we use machine learning?

- Complex patterns: When relationships in data are too complex for traditional programming and we are not able to find the patterns ourselves.
- Large datasets: When there is an abundance of data that can be leveraged for learning.

Remark. Why do we use it? Because it can generalize well to unseen data, making accurate predictions or decisions based on learned patterns (not memorize only).

Warning

Machine learning is as good as the data it is trained on. Poor quality or biased data can lead to inaccurate or unfair outcomes. In some domains, generalization is critical (e.g., medical diagnosis), while in others, memorization may suffice (e.g., recommendation systems).

1.5 Mechanics of machine learning

- Data Representation: Turn everything into features (numbers)
- Pattern discovery: Data looks random at the start, but then the structure emerges.
- Pick up a model: Line or something more complex.

- Training: This is an optimization problem. We define a loss function (error measure) and minimize it.
- Generalize: A good model should generalize well to unseen data (not just memorize training data).
- Evaluation: Assess model performance on unseen data using different metrics (accuracy, precision, etc.). This is the only performance measure that matters.

1.6 Data splitting

Definition 2. simeqentional data split:

- Training set: 70% of the data used to train the model.
- Validation set: 15% of the data used to tune hyperparameters and select the best model.
- Test set: 15% of the data used to evaluate the final model's performance.

Sometimes, we combine training and validation sets.

Caution

The test data is **never** used during training or model selection. It is only used for final evaluation to get an unbiased estimate of model performance.

Remark. For training, validation, and testing data, we known the features $(F_1, F_2, \ldots, F_n)$ and the labels (outputs) (Y). For unseen data, we only know the features and want to predict the labels. The key is generalization: the model should perform well on unseen data, not just the training data.

1.7 Types of Error:

- Training error: Error on the training set. It measures how well the model fits the training data.
- Validation error: Error on the validation set. It helps in tuning hyperparameters and selecting the best model.

- Test error: Error on the test set. It provides an unbiased estimate of the model's performance on unseen data.

1.8 Training Dynamics: Error vs Epochs

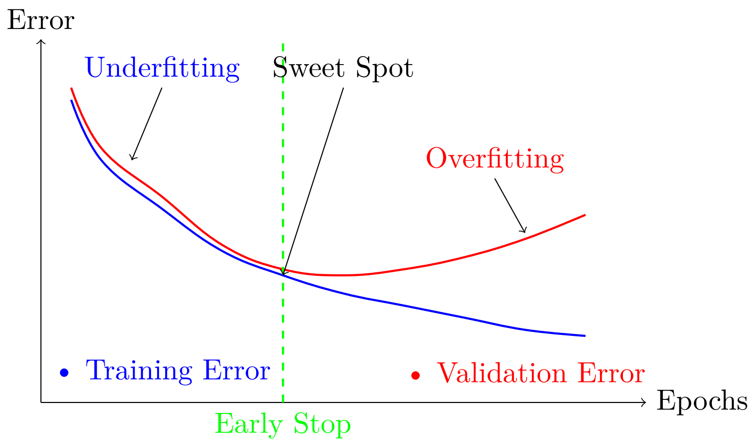


Figure 1: Training and validation error vs epochs showing underfitting, overfitting, and optimal early stopping point

Remark.

- **Underfitting**: Both training and validation errors are high. The model is too simple to capture the underlying patterns.
- **Overfitting**: Training error continues to decrease while validation error starts increasing. The model memorizes training data but fails to generalize.
- **Early Stopping**: Stop training when validation error starts increasing to prevent overfitting and achieve the best generalization.

1.8.1 Underfitting

- Model is too simple (high bias), Increase complexity
- Fails to capture underlying patterns
- Both training and validation errors are high

- More or better features may be needed
- More or better training data may be needed

1.8.2 Overfitting

- Get more data
- Regularization
- Drop out
- Batch normalization
- Data augmentation
- Early stopping

2 Tutorial 1: Overfitting and NumPy refresher

I missed this Tutorial but here is the link for reference: `https://www.dataquest.io/cheat-sheet/numpy-cheat-sheet/`

3 Lecture 2: Types of machine learning

3.1 Supervised learning

Definition 3. In supervised learning, the model is trained on a labeled dataset, where each input data point is paired with the correct output (label) $f : X \to Y$. The goal is to learn a mapping from inputs to outputs that can generalize to unseen data. We have labeled data: (X_i, Y_i) for $i = 1, 2, \ldots, N$, where X_i are the features and Y_i are the labels.

Example.
- Classification: Predicting discrete class labels (e.g., spam vs. not spam, digit recognition)
  - Binary classification: Two classes (e.g., spam vs. not spam)
  - Multi-class classification: More than two classes (e.g., digit recognition 0-9)
  - Multi-label classification: Each instance can belong to multiple classes simultaneously (e.g., tagging images with mul-

tiple labels)

- Regression: Predicting continuous outputs (e.g., house prices, temperature forecasting, MSE, MAE)
  - Linear regression
  - Polynomial regression
  - etc.

3.2 Unsupervised learning

Definition 4. In unsupervised learning, the model is trained on an unlabeled dataset, where the goal is to cluster in similar groups or find hidden patterns in the data. Another example is dimensionality reduction, where we reduce the number of features while preserving important information. We can also use it for anomaly detection, identifying rare or unusual data points that deviate significantly from the norm. There are no labels provided for the data points. We have unlabeled data: X_i for $i = 1, 2, \ldots, N$.

Example.

- Clustering: Grouping similar data points together (e.g., customer segmentation, image segmentation)
- Dimensionality reduction: Reducing the number of features while preserving important information (e.g., PCA, t-SNE)
- Anomaly detection: Identifying rare or unusual data points that deviate significantly from the norm (e.g., fraud detection, network security)

3.3 Reinforcement learning

Definition 5. In reinforcement learning, an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties based on its actions and aims to maximize cumulative rewards over time. The agent learns a policy that maps states to actions to achieve the best long-term outcome. Key components:

- Agent: The learner or decision-maker.

- Environment: The external system the agent interacts with.
- State: The current situation of the agent in the environment.
- Action: The choices available to the agent.
- Reward: Feedback signal indicating the success of an action.
- Policy: A strategy that defines the agent's actions based on the current state.

Example.
- Game playing: Training agents to play games like chess, Go, or video games (e.g., AlphaGo, DQN)
- Robotics: Teaching robots to perform tasks through trial and error (e.g., robotic arm manipulation)
- Autonomous vehicles: Enabling self-driving cars to navigate and make decisions in real-time (e.g., Tesla Autopilot)

3.4 Semi-supervised learning and self-supervised learning

Definition 6. Semi-supervised learning combines a small amount of labeled data with a large amount of unlabeled data during training. The model leverages the labeled data to learn initial patterns and then uses the unlabeled data to refine its understanding and improve generalization and performance.

Definition 7. Self-supervised learning is a type of unsupervised learning where the model generates its own labels from the input data.

Note. Integrating Mechanistic Models with Machine Learning: Using physical laws or domain knowledge to inform and constrain machine learning models, enhancing their interpretability and reliability.

Let's start with a supervised learning and classification example.

3.4.1 Classification

Problem: The input (feature vector) $x \in \mathbb{R}^d$ and the output (class label) $y \in \{1, \ldots, K\}$. We have a dataset of N samples: $\{(x_i, y_i)\}_{i=1}^N$. The goal is, given a new input x_{new}, predict its class label y_{new}.

A probabilistic vew of classification: We want to model the conditional probability (Posterior probability) $P(y|x)$, which gives the probability of each class label given the input features. We want to minimize the probability of misclassification.

Definition 8. **Bayes Decision Rule:** Given the posterior probabilities $P(y = k|x)$ for each class k, the Bayes classifier assign the input x to the class with the highest posterior probability:

$$\hat{y} = \arg\max_k P(y = k|x)$$

Remark. Bayes theorem:

$$P(y = k|x) = \frac{P(x|y = k)P(y = k)}{P(x)}$$

This implies that,

$$\hat{y} = \arg\max_k P(x|y = k)P(y = k)$$

Where,

- $P(x|y = k)$ is the likelihood of observing the features x given class k. This is class-conditional density.
- $P(y = k)$ is the prior probability of class k.

3.4.2 Linear Discriminant Analysis (LDA):

It is based on these assumptions:

- $P(x|y = k) = N(\mu_k, \Sigma_k)$ (multivariate Gaussian distribution with class-specific mean μ_k and covariance Σ_k)
- $\Sigma_k = \Sigma$ (shared covariance across classes)
- $\Pi_k = P(y = k)$ (Prior probabilities) μ_k : mean vector for class k
  This implies,

  $$P(x|y = k)p(y = k) = P(x|y = k)\Pi_k \implies \hat{y}(x) = \arg\max_k P(x|y = k)\Pi_k$$

Now let's work with the log of the above expression:

$$\begin{aligned}\tilde{y}(x) &= \arg\max_k \log(P(x|y=k)\Pi_k) \\ &= \arg\max_k \log(P(x|y=k)) + \log(\Pi_k)\end{aligned}$$

Doing some algebra, we get:

$$P(x|y=k) = \frac{1}{(2\pi)^d|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)\right)$$

Now, taking the log:

$$\begin{aligned}\log(P(x|y=k)) &= -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log(|\Sigma|) - \frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k) \\ &= -\frac{1}{2}x^T\Sigma^{-1}x + x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \frac{1}{2}\log(|\Sigma|)\end{aligned}$$

Thus substituting back into the expression for $\tilde{y}(x)$ and dropping the terms independent of k, we get:

$$\begin{aligned}\tilde{y}(x) &= \arg\max_k \left(-\frac{1}{2}x^T\Sigma^{-1}x + x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log(\Pi_k)\right) \\ &= \arg\max_k \left(x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log(\Pi_k)\right)\end{aligned}$$

Let,

$$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log(\Pi_k)$$

Thus, the decision rule becomes:

$$\tilde{y}(x) = \arg\max_k \delta_k(x)$$

Now suppose the case of two classes, i.e., $K = 2$,

$$\delta_1(x) - \delta_2(x) \geq 0$$

We substitute the expression for $\delta_k(x)$ and do some algebra to get:

$$w^Tx + b \geq 0$$

Where,

$$w = \Sigma^{-1}(\mu_1 - \mu_2)$$
$$b = -\frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \log\left(\frac{\Pi_1}{\Pi_2}\right)$$

This is a linear decision boundary, hence the name Linear Discriminant Analysis (LDA).

3.4.3 Logistic Regression:

It is a supervised learning algorithm used for binary classification tasks. It models the probability of the positive class (class 1) using the logistic (sigmoid) function. We approximate the posterior probability $P(y = 1k|x)$,

Problem: Binary classification with input features $x \in \mathbb{R}^d$ and output labels $y \in \{0, 1\}$. We have a dataset of N samples: $\{(x_i, y_i)\}_{i=1}^N$. The goal is to model the probability that an observation belongs to class 1 given its features given its features, $P(y = 1|x)$.

Recall that in linear regression, we model the output as a linear combination of input features:

$$\beta^T x = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_{d-1} x_{d-1}$$

Here,

$$\begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_{d-1} \end{pmatrix} = x \in \mathbb{R}^d, \quad \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{d-1} \end{pmatrix} \in \mathbb{R}^d$$

Info

Here when you expand $\beta^T x$, the first term β_0 is the intercept (bias term), and the remaining terms $\beta_1 x_1, \beta_2 x_2, \ldots, \beta_{d-1} x_{d-1}$ are the contributions from each feature weighted by their respective coefficients.

And,

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

Where $\sigma(z)$ is the sigmoid function that maps any real-valued number to the (0, 1) interval, making it suitable for modeling probabilities.

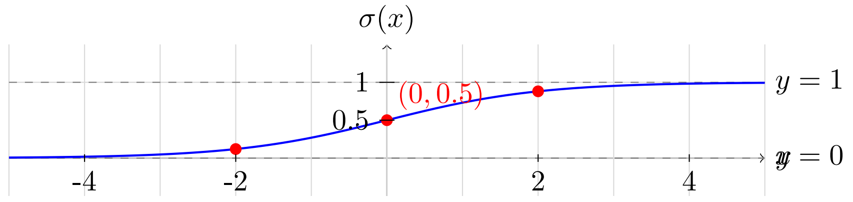


Figure 2: The sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$

So we have that,

$$P(y=1|x) = \sigma(\beta^T x) = \frac{1}{1+e^{-\beta^T x}}$$

Thus,

$$P(y=0|x) = 1 - P(y=1|x) = 1 - \sigma(\beta^T x) = \frac{e^{-\beta^T x}}{1+e^{-\beta^T x}}$$

Likehood function: Given the dataset $\{(x_i, y_i)\}_{i=1}^N$ and $y \in (0,1)$, the likelihood function $L(\beta)$ is the product of the probabilities of observing each data point:

$$P(y_i|x_i,\beta) = (P(y=1|x_i))^{y_i}(P(y=0|x_i))^{1-y_i}$$

$$L(\beta) = \prod_{i=1}^N P(y_i|x_i,\beta) = \prod_{i=1}^N (\sigma(\beta^T x_i))^{y_i}(1-\sigma(\beta^T x_i))^{1-y_i}$$

We work with the log-likelihood function:

$$\begin{aligned}\ell(\beta) = \tilde{L(\beta)} = \log(L(\beta))\\ = \sum_{i=1}^N \left[y_i \log\Big(\sigma(\beta^T x_i)\Big) + (1-y_i)\log\Big(1-\sigma(\beta^T x_i)\Big)\right]\end{aligned}$$

4 Lecture 3: Gradient Descent and Logistic Regression Examples

4.1 Optimization of the Likelihood Function

Recall from the previous lecture that we derived the log-likelihood function $\tilde{L}(\beta)$ for logistic regression. We want to find the parameters β that maximize this likelihood.

Definition 9. **Binary Cross Entropy Loss:** Maximizing the log-likelihood is equivalent to minimizing the negative log-likelihood. We define the cost function (or loss function) $J(\beta)$ as:

$$J(\beta) = -\tilde{L}(\beta) = -\sum_{i=1}^{N} \left[y_i \log\Big(\sigma(\beta^T x_i)\Big) + (1 - y_i) \log\Big(1 - \sigma(\beta^T x_i)\Big) \right]$$

Our optimization problem is:

$$\min_{\beta} J(\beta)$$

Remark. Since $J(\beta)$ is non-linear (due to the sigmoid function $\sigma(z)$), there is no closed-form solution (unlike Linear Regression where we had the Normal Equations). Therefore, we must use iterative optimization methods.

4.2 Method of Gradient Descent

Definition 10. **Gradient Descent:** An iterative optimization algorithm for finding the local minimum of a differentiable function. Suppose we want to find β^* such that $\nabla f(\beta^*) = 0$ for a function $f : \mathbb{R}^d \to \mathbb{R}$.
The directional derivative is minimized in the direction of the negative gradient. The update rule is:

$$\beta^{(k+1)} = \beta^{(k)} - \eta \nabla f(\beta^{(k)})$$

Where:

- $\beta^{(k)}$ is the parameter vector at step k.
- η is the **learning rate** (step size), which can be fixed or adaptive.
- $\nabla f(\beta^{(k)})$ is the gradient of the loss function at the current step.

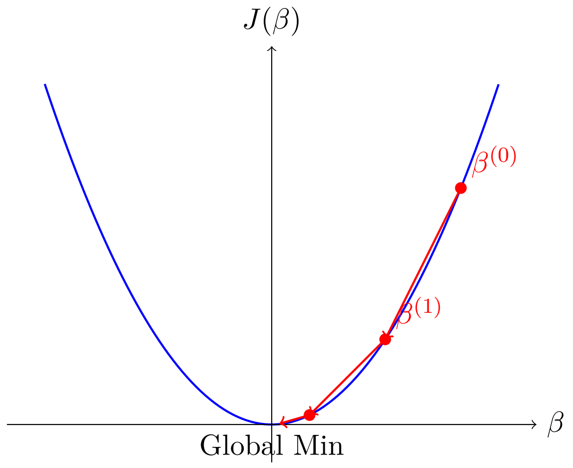


Figure 3: Visualizing Gradient Descent: Taking steps proportional to the negative of the gradient to reach the minimum.

4.2.1 Deriving the Gradient for Logistic Regression

We need to calculate $\nabla_\beta J(\beta)$. Let $z_i = \beta^T x_i$. We use the following properties of the sigmoid derivative:

$$\frac{d}{dz}\log\sigma(z) = 1 - \sigma(z) \quad \text{and} \quad \frac{d}{dz}\log(1-\sigma(z)) = -\sigma(z)$$

Also note that $\nabla_\beta z_i = x_i$.

Applying the chain rule to our loss function terms:

$$\nabla_\beta[y_i \log(\sigma(z_i))] = y_i(1-\sigma(z_i))x_i$$
$$\nabla_\beta[(1-y_i)\log(1-\sigma(z_i))] = -(1-y_i)\sigma(z_i)x_i$$

Summing these up for the full gradient:

$$\begin{aligned}\nabla_\beta J(\beta) &= -\sum_{i=1}^{N} [y_i(1-\sigma(z_i))x_i - (1-y_i)\sigma(z_i)x_i] \\ &= -\sum_{i=1}^{N} [(y_i - y_i\sigma(z_i) - \sigma(z_i) + y_i\sigma(z_i))x_i] \\ &= -\sum_{i=1}^{N} (y_i - \sigma(z_i))x_i \\ &= \sum_{i=1}^{N} (\sigma(\beta^T x_i) - y_i)x_i\end{aligned}$$

Thus, the gradient descent update rule for logistic regression is:

$$\beta^{(k+1)} = \beta^{(k)} - \eta \sum_{i=1}^{N} (\sigma(\beta^{(k)T} x_i) - y_i)x_i$$

4.3 Examples of Logistic Regression

4.3.1 Example: detecting phase transition in Ising Model using logistic regression

Example. We start with a 2D lattice where each node has a spin, either up or down. If the temperature is above a critical temperature T_c the spins are in disorder, but when the temperature drops below T_c the spins are in order.
The Hamiltonian is $H = -J\sum_{\langle ij \rangle} \sigma_i \sigma_j$ where $\langle ij \rangle$ refers to nearest neighbour interaction 82].

$$\begin{aligned} T > T_c &\quad \text{disorder} \\ T < T_c &\quad \text{order} \end{aligned}$$

Data (configurations) can be generated using monte carlo (which will be provided in assignments). If the temperature is far from T_c, it will be easy to classify. If it is close to T_c, it will be difficult 85]. The classification won't be able to find an exact T_c, just a range.
Geometrically, this is a linear classification problem but with nonlinear dynamics (?), so a simple model doesn't work well. SVD, logistic regression, and decision trees are all linear regression methods which will not work well for this problem. Neural networks work much better,

as we will see in assignment 2.

4.3.2 Example: prediction of LORIS immunotherapy response

Example. Can we predict, prior to treatment, the probability that a patient will respond to immunotherapy using measured clinical variance?
Problem: we have labelled data for 6 features $x \in \mathbb{R}^6$ with n labelled data points $\{(x_i, y_i)\}_{i=1}^n$. We want to use logistic regression, which turns out to be an even better method than neural networks. Let $P(y = 1|x)$ be the probability of response.

$$\begin{aligned} P(y=1|x) &= \sigma(\beta^T x_i) \\ &= \sigma\left(\beta_0 + \sum_{j=1}^{6} \beta_j x_j\right) \end{aligned}$$

Here, β_0 is the bias term.
The log-odds representation is a key advantage of logistic regression. We have the odds of response:

$$\frac{P(y=1|x)}{P(y=0|x)} = \frac{P(y=1|x)}{1 - P(y=1|x)}$$

Take the log of both sides.

$$\log \frac{P(y=1|x)}{P(y=0|x)} = \beta_0 + \sum_{j=1}^{6} \beta_j x_j$$

4.4 Log-Odds of Response Approach

An equivalent and often more interpretable form of logistic regression is obtained by considering the odds and log-odds of response. The odds of response given the clinical features $\mathbf{X}$ are defined as

$$\frac{\mathbb{P}(y=1 \mid \mathbf{X})}{\mathbb{P}(y=0 \mid \mathbf{X})} = \frac{\mathbb{P}(y=1 \mid \mathbf{X})}{1 - \mathbb{P}(y=1 \mid \mathbf{X})}.$$

Using the logistic regression model

$$\mathbb{P}(y = 1 \mid \mathbf{X}) = \sigma\left(\beta_0 + \sum_{j=1}^{6} \beta_j X_j\right),$$

we obtain the log-odds:

$$\log \frac{\mathbb{P}(y = 1 \mid \mathbf{X})}{\mathbb{P}(y = 0 \mid \mathbf{X})} = \log \frac{\mathbb{P}(y = 1 \mid \mathbf{X})}{1 - \mathbb{P}(y = 1 \mid \mathbf{X})} \tag{1}$$

$$= \beta_0 + \sum_{j=1}^{6} \beta_j X_j. \tag{2}$$

5 Lecture 4: Support Vector Machines (SVMs)

5.1 Introduction to Support Vector Machines

Definition 11. Support Vector Machines (SVMs) are supervised learning models ($\{x_i, y_i\}_{i=1}^n$) used for classification ($y_i = \{+1, -1\}$) and regression tasks. They work by finding the optimal line or hyperplane that separates data points of different classes. So, if we have two classes that are perfectly linearly separable, we can find a hyperplane that separates them with the maximum margin.

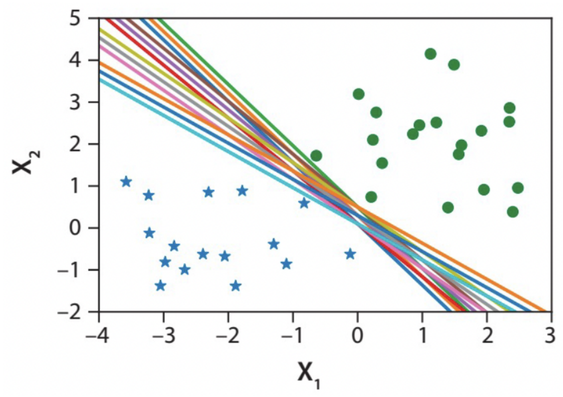


Figure 4: SVM finding the optimal hyperplane with maximum margin

5.1.1 Decision Boundary

The Classifer is defined as:

$$f(x) = w^T x + b \qquad \text{sign}(f(x))$$

Where:

- w is the the normal vector to the decision boundary (hyperplane)
- b is the bias term (offset from origin)
- The decision boundary is defined by $f(x) = 0$

5.1.2 Properties

Property 1. The decision boundary (hyperplane) has the following properties:

- For any two points x_1 and x_2 on the hyperplane:

$$\begin{cases} w^T x_1 + b = 0 \\ w^T x_2 + b = 0 \end{cases} \implies w^T(x_1 - x_2) = 0 \implies w \perp (x_1 - x_2)$$

- Distance from point x_0 to hyperplane:
  If x_0 is on the decision boundary, then

$$w^T x_0 + b = 0 \implies b = -w^T x_0 \implies f(x) = w^T(x - x_0)$$

  The distance d from point x to the hyperplane is given by the projection of the vector $(x - x_0)$ onto the normal vector w:

$$\text{Dis}(x, H) = \frac{f(x)}{\|w\|} = \frac{w^T x + b}{\|w\|}$$

  H is the hyperplane/dicision boundary.

- For a correctly classified point (x_i, y_i):

$$y_i f(x_i) = y_i(w^T x_i + b) = 1 \implies \text{margin} \begin{cases} f(x_i) \geq 1 & \text{if } y_i = +1 \\ f(x_i) \leq -1 & \text{if } y_i = -1 \end{cases}$$

5.2 Maximizing the Margin

Definition 12. The margin is the distance between the decision boundary and the closest data points from each class (support vectors). The goal of SVM is to find the hyperplane that maximizes this margin, leading to better generalization on unseen data.

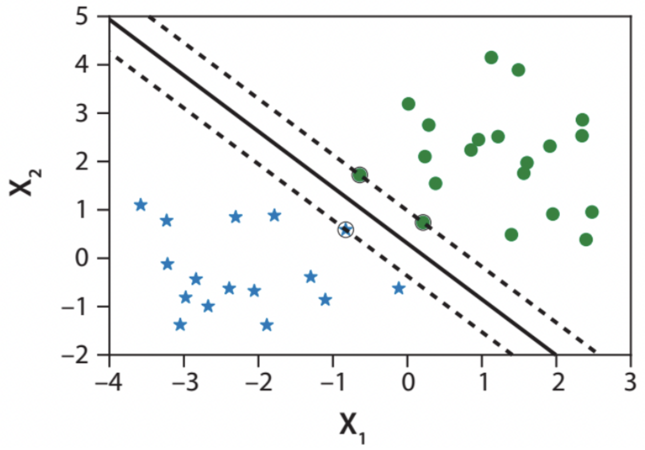


Figure 5: Maximizing the margin between classes

So, we have the margin defined as:

$$\text{Margin} = \frac{1}{\|w\|} + \frac{|-1|}{\|w\|} = \frac{2}{\|w\|}$$

The goal of SVM is to maximize the margin,

$$\max_{w,b} \frac{2}{\|w\|} \equiv \min_{w,b} \frac{1}{2}\|w\|^2$$
$$\text{subject to } y_i(w^T x_i + b) \geq 1, \quad i = 1, 2, \ldots, N$$

Or equivalently,

$$\min_{w,b} \frac{1}{2}\|w\|^2$$
$$\text{subject to } y_i(w^T x_i + b) - 1 \geq 0, \quad i = 1, 2, \ldots, N$$

We can use Lagrange multipliers to solve this constrained optimization problem.

Definition 13. **Lagrangian:** The Lagrangian function for the SVM optimization problem is defined as:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{N} \alpha_i [y_i(w^T x_i + b) - 1]$$

Where $\alpha_i \geq 0$ are the Lagrange multipliers associated with each constraint.

To find the optimal solution, we take the partial derivatives of the Lagrangian with respect to w and b, set them to zero, and solve for w and b:

$$\frac{\partial \mathcal{L}}{\partial w} = w - \sum_{i=1}^{N} \alpha_i y_i x_i = 0 \implies w = \sum_{i=1}^{N} \alpha_i y_i x_i$$
$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{N} \alpha_i y_i = 0 \implies \sum_{i=1}^{N} \alpha_i y_i = 0$$

Now, we have the condition:

$$\alpha_i(y_i(w^T x_i + b) - 1) = 0 \implies \alpha_i = 0 \text{ or (when } a_i > 0) \; y_i(w^T x_i + b) - 1 = 0$$

Info

Notice that this is implies that only the support vectors (points closest to the decision boundary) will have non-zero α_i. These points are critical in defining the position and orientation of the hyperplane. So, only certain data points (support vectors) influence the final model.

Definition 14.

$$w = \sum_{i=1}^{N} \alpha_i y_i x_i$$

are the support vectors.

We can now apply the dual optimization to find the optimal α_i.

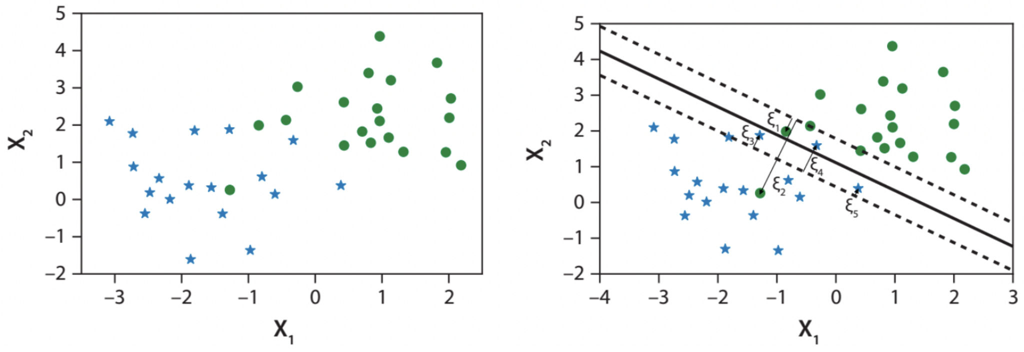


Figure 6: Soft-Margin SVM allowing misclassifications

The dual optimization problem is:

$$\max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$\text{subject to } \alpha_i \geq 0, \quad i = 1, 2, \ldots, N$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

Note. This is called the **Hard-Margin SVM**.

5.3 Soft-Margin SVM

In real-world scenarios, data is often not perfectly linearly separable. To handle such cases, we introduce slack variables $\xi_i \geq 0$ to allow some misclassifications while still trying to maximize the margin. So, our decision boundary constraints become:

$$\begin{aligned} y_i(w^T x_i + b) &\geq 1 - \xi_i, \quad i = 1, 2, \ldots, N \\ \xi_i &\geq 0, \quad i = 1, 2, \ldots, N \end{aligned}$$

Remark. Notice that if $\xi_i = 0$, the point is correctly classified and outside the margin. If $0 < \xi_i < 1$, the point is inside the margin but still correctly classified. If $\xi_i > 1$, the point is misclassified (on the wrong side of the decision boundary).

5.3.1 Optimization Problem for Soft-Margin SVM

The optimization problem for Soft-Margin SVM is formulated as:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i$$

$$\text{subject to } y_i(w^T x_i + b) \geq 1 - \xi_i, \quad i = 1, 2, \ldots, N$$

$$\xi_i \geq 0, \quad i = 1, 2, \ldots, N$$

Property 2. Here, $C > 0$ is a regularization parameter that controls the trade-off between maximizing the margin and minimizing the classification error. A larger C puts more emphasis on minimizing misclassifications (fewer training errors but potentially a smaller margin), while a smaller C allows for a wider margin with potentiallymore misclassifications and better generalization.

5.4 Non-Linear SVM

5.4.1 Kernel Trick

What about non-linear decision boundaries? We can use the **Kernel Trick** to map the input data into a higher-dimensional feature space where a linear decision boundary can be found.

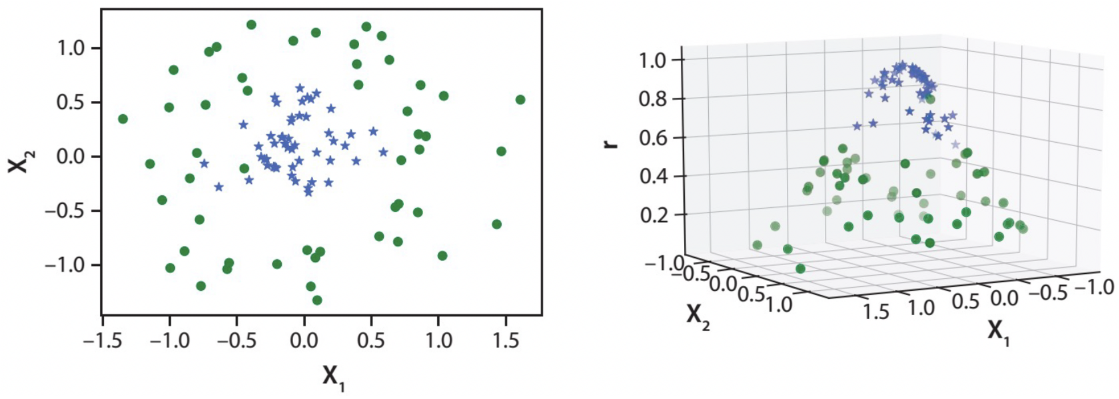


Figure 7: Using the Kernel Trick to handle non-linear decision boundaries

Review 1. In the optimization problem for the linear case, we had,

$$\frac{1}{2}\|w\|^2 + \dots$$

$$w = \sum_{i=1}^{N} \alpha_i y_i x_i$$

The dual optimization is,

$$\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

In the non-linear case, we replace the dot product $x_i^T x_j$ with a kernel function $K(x_i, x_j)$ that computes the inner product in the transformed feature space without explicitly performing the transformation.

$$x \to \phi(x)$$
$$x_i^T x_j \to K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

Some common kernel functions include:

- Linear Kernel: $K(x_i, x_j) = x_i^T x_j$
- Polynomial Kernel: $K(x_i, x_j) = (x_i^T x_j + c)^d$
- Gaussian Kernel: $K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$

5.4.2 Examples

Example. **Handwritten digit classification**.
Our dataset consists of images of handwritten digits (0-9) (e.g., MNIST dataset). Now each image is 28x28 pixels (Gray scale), so each image can be represented as a 784-dimensional vector $x \in \mathbb{R}^{784}$ (We are flattening the 2D image into a 1D vector). The goal is to classify each image into one of the 10 digit classes (0-9).

Binary classification: Let say for exameple we want classify between the digits:

- digits 0 and 1 (easier task)
- digits 3 and 8 (harder task)

For Case 1, the data are almost linearly separable in $\mathbb{R}^{784}$ space, so a linear SVM works and gives a high accuracy. For Case 2, the data is not linearly separable, so we can use a non-linear SVM to achieve better classification performance.

Multi-class classification: We can extend SVM to multi-class classification using a tree of binary classifiers (one-vs-one or one-vs-all approach). Each binary classifier is trained to distinguish between two classes, and the final prediction is made based on the outputs of all classifiers.

Example. This example is based on the following papaer: Exotic and physics-informed support vector machines for high energy physics **Drell-Yan event classification**.
Proton-Proton Collisons:

$$PP \to Z \to l^+ l^-$$

Where l is a lepton (electron or positron) and Z is a boson heavy particle.

From the observed kinetics (energy, momentum, angles, etc.) of the leptons, we want to classify whether the event comes from a Z boson decay (signal) or from other sources (background).

Goal: Apply a non-linear SVM with a physics-informed kernel (instead of polynomial or Gaussian kernels) to classify Drell-Yan events.

Drell-Yan dynamics depends on the invariant mass M_z, momentum fraction x_1, x_2 and repidity $\frac{1}{2}\ln(x_1/x_2) = y$. So, we can define a physics-informed kernel as:

$$K(x,z) = \gamma(\langle x|z\rangle^2 + \langle x|z\rangle + \langle x|z\rangle\, e^{\langle x|z\rangle})$$

Where $\langle x|z\rangle^2$ is M_z^2, $\langle x|z\rangle$ is M_z and $\langle x|z\rangle\, e^{\langle x|z\rangle}$ is $M_z e^{\pm y}$.

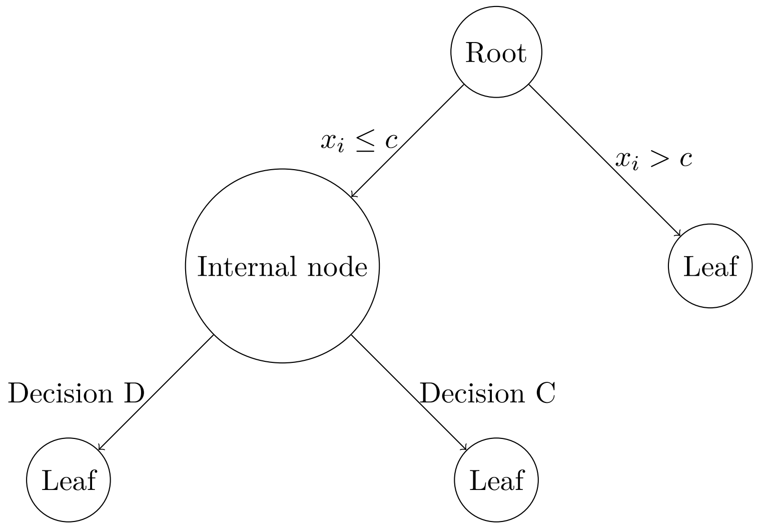
6 Lecture 5: Decision Trees and Random Forests

6.1 Building Decision Trees

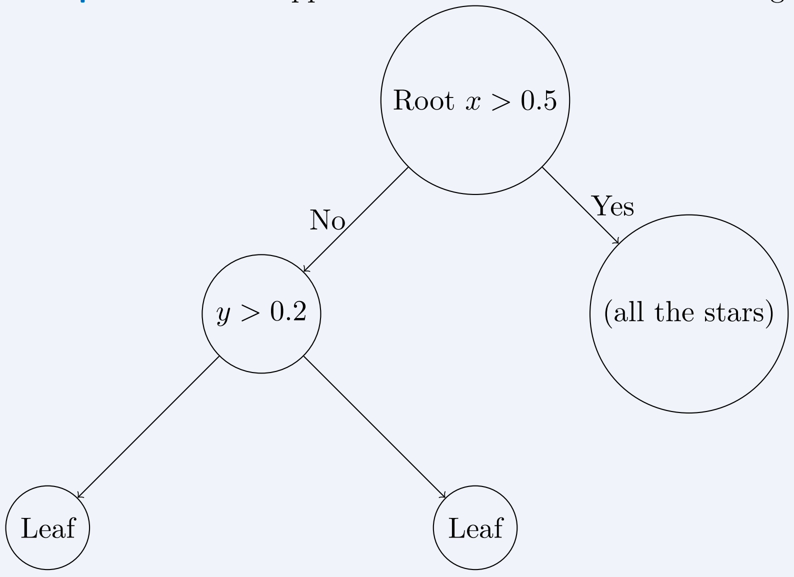
Definition 15. Decision Trees are supervised learning models used for classification and regression tasks. They work by recursively partitioning the the data using simple if-then rules based on feature values, resulting in a tree-like structure where each internal node represents a decision based on a feature, each branch represents the outcome of that decision, and each leaf node represents a class label (for classification) or a continuous value (for regression).

Property 3. The basic structure of a decision tree consists of:

- Root Node: The topmost node representing the entire dataset.
- Internal Nodes: Nodes that represent decisions based on feature values.
- Leaf Nodes: Terminal nodes that represent the final output (class label or value).



Example. Suppose we have the following tree:



Note. Here we can also start with $y > 0.2$ and then $x > 0.5$, which will give a different tree structure but the same partitioning of the feature space.

This leads to the following question: Which split should we apply first?

6.1.1 Gini Impurity

To decide which feature to split on at each node, we can use the Gini Impurity measure.

Definition 16. **Gini Impurity:** Consider a node contanining a sample from C classes. Let p_i be the probability of belonging to class i in that node.

$$\sum_{i=1}^{C} p_i = 1$$

Now suppose we assign a random label according to data distribution (i.e., with probability p_i for class i). If the true class is i, the probability of misclassification is $1 - p_i$. Now the joint probability of choosing class i and misclassifying it is,

$$p_i(1-p_i) = p_i - p_i^2$$

So, the total probability of misclassification (Gini Impurity) is,

$$G = \sum_{i=1}^{C} p_i(1-p_i) = \sum_{i=1}^{C}(p_i - p_i^2) = 1 - \sum_{i=1}^{C} p_i^2$$

In the binary classification case ($C = 2$), if p is the probability of class 1, then the Gini Impurity simplifies to:

$$G = 2p(1-p)$$

Example. Now going back the example of spliting the dots and stars, We had 6 dots, 9 stars. So let p be the probability of a dot, then

$$p = \frac{6}{15} = 0.4 \quad 1-p = \frac{9}{15} = 0.6$$

So the Gini Impurity is,

$$G = 2p(1-p) = 2 \times 0.4 \times 0.6 = 0.48$$

Now lets consider the horizontal split:

$$G_{bottom} = 0 (4 \text{ stars only, pure})$$

$$G_{top} = 1 - \left(\frac{6}{11}\right)^2 - \left(\frac{5}{11}\right)^2 = 0.4959$$

The weighted Gini Impurity is,

$$G_{after} = \frac{4}{15} \times 0 + \frac{11}{15} \times 0.4959 = 0.3637$$

So,

$$\Delta G = G_{initial} - G_{after} = 0.48 - 0.3637 = 0.1163$$

Similarly for the vertical split:

$$G_{right} = 0$$

$$G_{left} = 1 - \left(\frac{2}{8}\right)^2 - \left(\frac{6}{8}\right)^2 = 0.375$$

The weighted Gini Impurity is,

$$G_{after} = \frac{7}{15} \times 0 + \frac{8}{15} \times 0.375 = 0.2$$

So,

$$\Delta G = G_{initial} - G_{after} = 0.48 - 0.2 = 0.28$$

Thus, the vertical split is better as it gives a larger decrease in Gini Impurity. So we should split on $x > 0.5$ first.

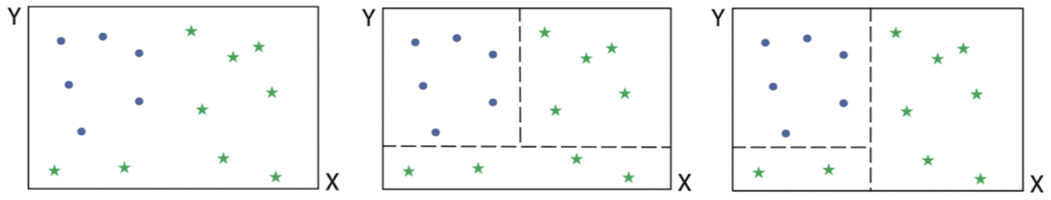


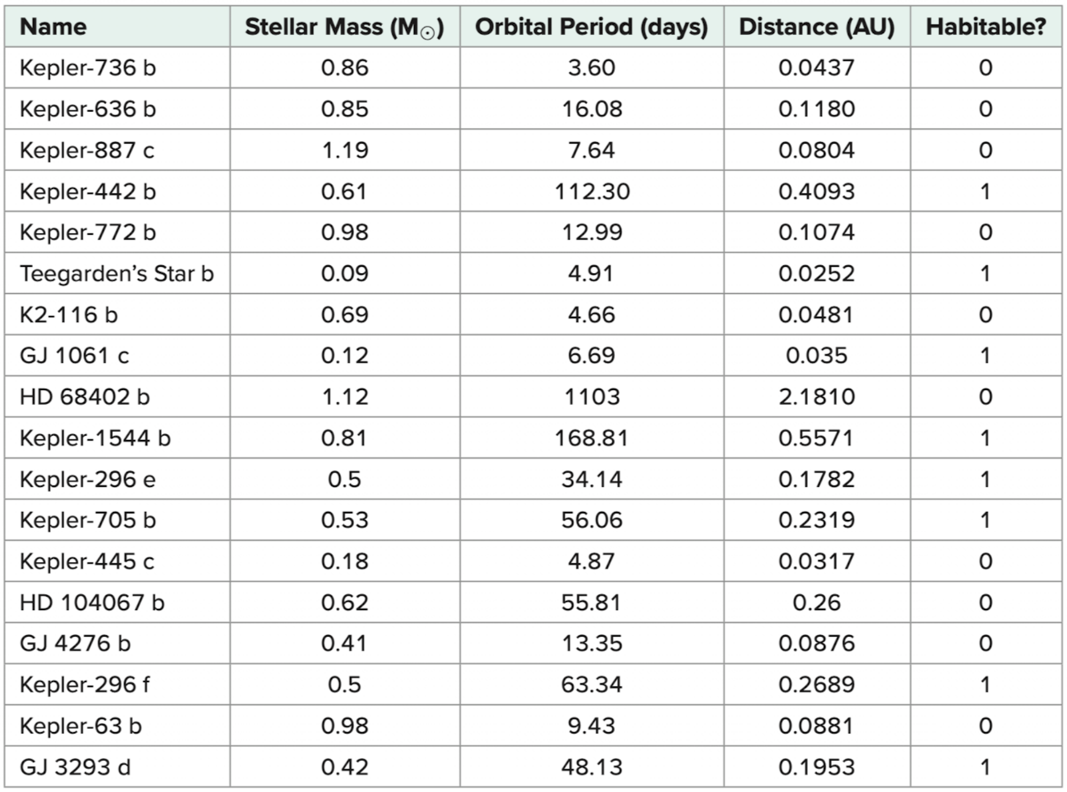
Figure 8: Decision tree example partitioning the feature space

6.2 When to Stop Splitting?

To prevent overfitting and ensure that the decision tree generalizes well to unseen data, we need to decide when to stop splitting nodes. Common stopping criteria include:

- Maximum Depth: Limit the depth of the tree to a predefined value.
- Minimum Samples per Leaf: Require a minimum number of samples in each leaf node.
- Minimum Impurity Decrease: Stop splitting if the decrease in impurity (e.g., Gini Impurity) is below a certain threshold.
- Pure Nodes: Stop splitting when all samples in a node belong to the same class.

We can also apply **pruning** techniques after the tree is fully grown (possible overfitting) to remove branches that do not provide significant predictive power.

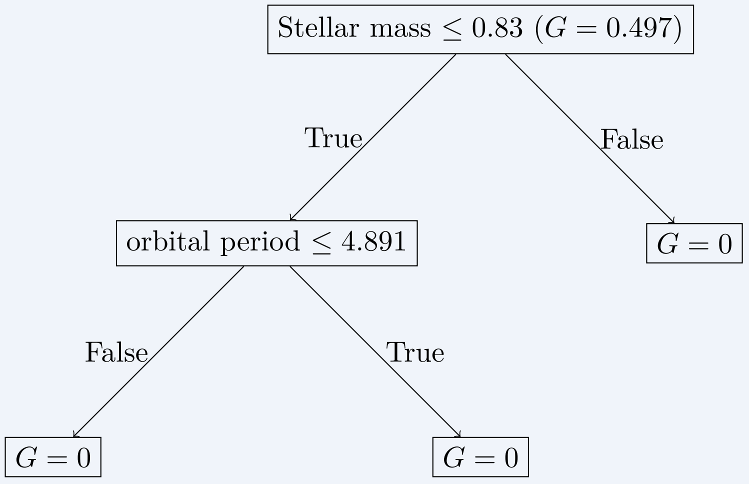


Name	Stellar Mass ($M_\odot$)	Orbital Period (days)	Distance (AU)	Habitable?
Kepler-736 b	0.86	3.60	0.0437	0
Kepler-636 b	0.85	16.08	0.1180	0
Kepler-887 c	1.19	7.64	0.0804	0
Kepler-442 b	0.61	112.30	0.4093	1
Kepler-772 b	0.98	12.99	0.1074	0
Teegarden's Star b	0.09	4.91	0.0252	1
K2-116 b	0.69	4.66	0.0481	0
GJ 1061 c	0.12	6.69	0.035	1
HD 68402 b	1.12	1103	2.1810	0
Kepler-1544 b	0.81	168.81	0.5571	1
Kepler-296 e	0.5	34.14	0.1782	1
Kepler-705 b	0.53	56.06	0.2319	1
Kepler-445 c	0.18	4.87	0.0317	0
HD 104067 b	0.62	55.81	0.26	0
GJ 4276 b	0.41	13.35	0.0876	0
Kepler-296 f	0.5	63.34	0.2689	1
Kepler-63 b	0.98	9.43	0.0881	0
GJ 3293 d	0.42	48.13	0.1953	1

Figure 9: Planet Habitability data

Example. Planet Habitability Classification.
So we have 10 habitable planets, 8 non-habitable planets and the outlier: HD68402b (long orbital period).
Data spliting:

- We ise the first 13 samples for training and the 5 remaining samples for testing.
- We use the last 13 samples for training and the first 5 samples for testing.
- We give perfect classification on training data data but only about 60% accuracy on test data.



We only need two features leafs are 100% pure.
We can make an even deeper tree:

- Deeper tree with up to 4 splits.
- All three features are used.
- The leafs nodes remain 100% pure but decision tress are highly sensitive to training data.

6.3 Random forests

Definition 17. Random Forests are ensemble learning methods that combine multiple decision trees to improve predictive performance and reduce overfitting. Each tree is trained on a random subset of the data and features, and the final prediction is made by aggregating the predictions from all trees (e.g., majority voting for classification or averaging for regression).

Property 4.
- Bootstrap Sampling: Generate n different training datasets by sampling with replacement from the original dataset. Each datatset has the same size as the original dataset but may contain duplicate samples.
- : Tree Construction: For each bootstrap sample, train a decision tree. At each node, randomly select k features from the total features m $(k < m)$ and choose the best split among those features (Gini impurity). We repeat this for each tree until a stopping criterion is met (e.g., maximum depth, minimum samples per leaf).
- Prediction Aggregation: For classification tasks, each tree in the forest makes a prediction, and the final prediction is determined by majority voting. For regression tasks, the final prediction is the average of the predictions from all trees.

The combination of bootstrap samplig and aggregation is known as bagging (bootstrap aggregating), which helps to reduce variance and improve the robustness of the model.

Remark. For those intrested in the paper mention the class for speeding up training in decision trees and random forests: Des-q: a quantum algorithm to provably speedup retraining of decision trees

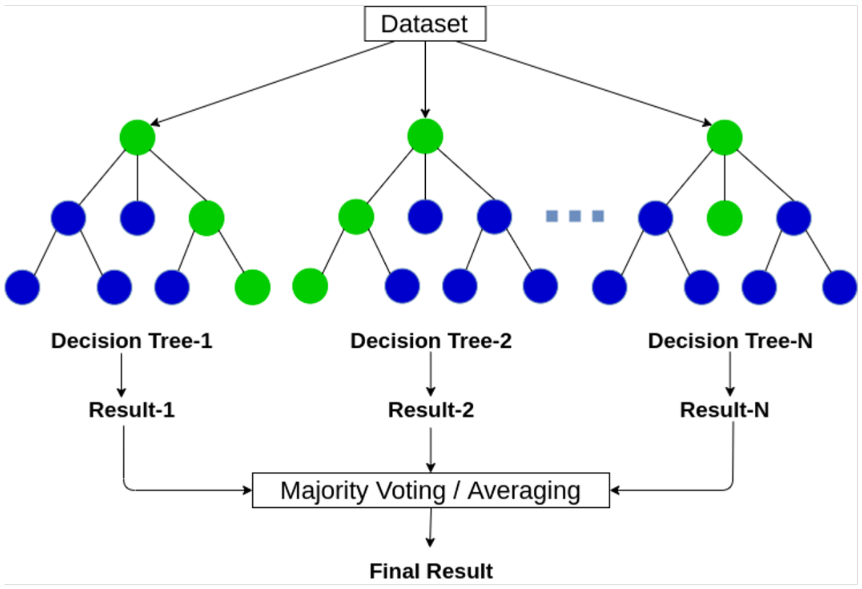


Figure 10: Random Forests: Combining multiple decision trees for improved performance

7 Lecture 6: K-Nearest Neighbors (KNN)

7.1 Introduction to K-Nearest Neighbors

Definition 18. K-Nearest Neighbors (KNN) is a simple, non-parametric, and instance-based supervised learning algorithm used for classification and regression tasks. The core idea of KNN is that data points that are close to each other in the feature space are likely to have similar labels or values. KNN makes predictions based on the labels or values of the k nearest neighbors in the training dataset.

Property 5. The KNN algorithm works as follows: Suppose we have the training dataset $\{(x_i, y_i)\}_{i=1}^N$, where $x_i \in \mathbb{R}^d$ is the feature vector and y_i is the corresponding label or value. Given a new data point x_{new}, we want to clasify it or predict its value.

- Compute the distance between x_{new} and all training data points x_i using a distance metric (e.g., Euclidean distance, Manhattan distance).

- Sort the distances and select the k nearest neighbors.
- Make a prediction based on the labels or values of the k nearest neighbors:
  - For classification: Use majority voting to determine the class label of x_{new}.
  - For regression: Calculate the average (or weighted average) of the values of the k nearest neighbors to predict the value for x_{new}.

$$\text{dist}(x_i, x_j) = \sqrt{\sum_{l=1}^{d}(x_{i,l} - x_{j,l})^2} \quad \text{(Euclidean distance)}$$

Remark. We can choose different values of k to control the model's complexity. A smaller k leads to a more flexible model that captures local patterns but may be sensitive to noise, while a larger k results in a smoother decision boundary but may overlook local variations.

7.1.1 Pros and Cons of KNN

- Pros:
  - Simple and easy to implement.
  - No training phase, making it fast for small datasets.
  - Can handle multi-class classification problems.
- Cons:
  - Computationally expensive for large datasets due to distance calculations.
  - Sensitive to irrelevant features and the choice of distance metric.
  - Requires careful selection of k and may suffer from the curse of dimensionality.

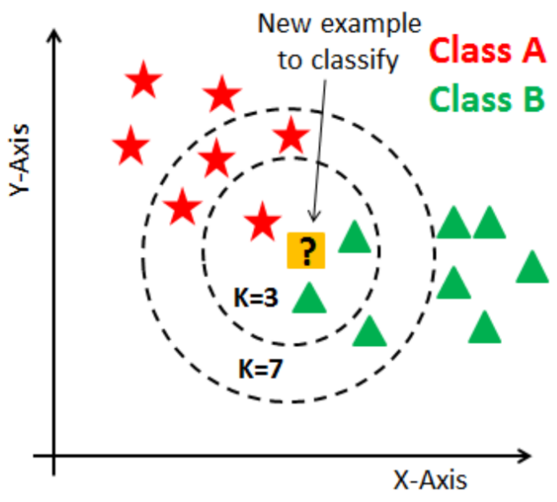


Figure 11: K-Nearest Neighbors: Classifying a new data point based on its nearest neighbors

7.1.2 Example

Going back to the habitable planet classification example (figure 9), we can use the KNN on the full dataset.

We have 4048 exoplanets with 97 numerical features, but we will only use 3 features: stellar mass, orbital period, and orbital distance. First, removing theose with missing values and outliers, we end up with 3171 planets.
Key: Most planets are non-habitables, so our data is imbalanced. We have only 52 habitable planets and 3119 non-habitable planets.

Before applying KNN, let's use a a lazy classifier that always predicts the majority class (non-habitable). This gives us an accuracy of:

$$\text{Accuracy} \approx \frac{3119}{3119 + 52} \approx 98.36\%$$

Definition 19. **Confusion Matrix**: A confusion matrix is a table used to evaluate the performance of a classification model. It summarizes the number of correct and incorrect predictions made by the model, broken down by each class. The confusion matrix for a binary classification problem typically has the following structure:

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

Where:

- TP (True Positive): The number of positive instances correctly predicted as positive.
- TN (True Negative): The number of negative instances correctly predicted as negative.
- FP (False Positive): The number of negative instances incorrectly predicted as positive.
- FN (False Negative): The number of positive instances incorrectly predicted as negative.
- Accuracy: The overall accuracy of the model is calculated as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- Precision: The precision of the model is calculated as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

- Recall: The recall of the model is calculated as:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Caution

Notice that accuracy can be misleading in imbalanced datasets. In our case, the model predicts all planets as non-habitable, resulting in a high accuracy of 98.36%, but it fails to identify any habitable planets (0% recall for the habitable class). This highlights the importance of considering other metrics like precision and recall when evaluating model performance, especially in imbalanced datasets.

For our lazy classifier, the confusion matrix is:

	Predicted non-habitable	Predicted habitable
Actual non-habitable	3119	0
Actual habitable	52	0

So, we have:

- Accuracy: $\frac{3119+0}{3119+0+0+52} \approx 98.36\%$
- Precision: $\frac{0}{0+0} =$ undefined (no positive predictions)
- Recall: $\frac{0}{0+52} = 0\%$

Remark. High Precision: Few false positives, but many missed positive cases (low recall).
High Recall with low Precision: Most positives are detected, but many false positives.

7.2 Cross-Validation

This is used for small or imbalanced datasets to better estimate model performance.

Definition 20. **K-Fold Cross-Validation**: In K-Fold Cross-Validation, the dataset is divided into k equal-sized folds (subsets). The model is trained k times, each time using $k-1$ folds for training and the remaining fold for validation. The performance metrics (e.g., accuracy, precision, recall) are averaged over the k iterations to provide a more robust estimate of the model's performance.

Remark. Each time we train the model on a different subset of the data, ensuring that every data point is used for both training and validation exactly once. This helps to mitigate overfitting and provides a better estimate of how the model will perform on unseen data.

KNN for planet habitability classification using 5-Fold Cross-Validation:

- They use 10-fold cross-validation.

- For $k = 5$, we get the following confusion matrix (averaged over 10 runs):

	Predicted non-habitable	Predicted habitable
Actual non-habitable	3097	22
Actual habitable	38	14

  They had about 50% accuracy on training data and about 30% on test data. This is a sign of underfitting.

- So, we have:
  - Accuracy: $\approx 98\%$
  - Precision: $\approx 32\%$
  - Recall: $\approx 27\%$

- Using decision tree, we get the following confusion matrix:

	Predicted non-habitable	Predicted habitable
Actual non-habitable	3092	27
Actual habitable	26	26

  They 100% accuracy on training data but only about 50% on test data. This is a sign of overfitting.

- So, we have:
  - Accuracy: $\approx 98.3\%$
  - Low Precision
  - Low Recall

The learning curves for both KNN and Decision Tree models are shown below:

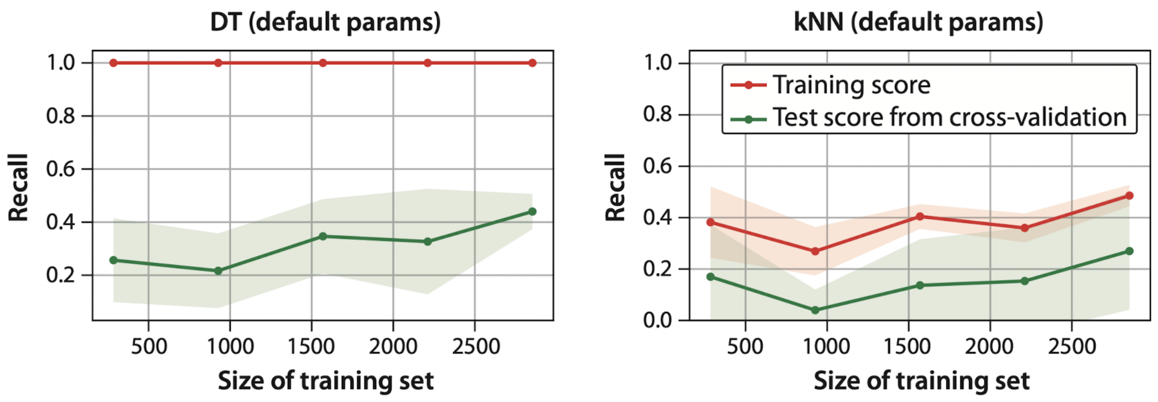


Figure 12: Learning Curves for Decision Tree (High Variance) and KNN (High Bias)

Definition 21. **Bias:** is the error due to overly simplistic assumptions in the learning algorithm. High bias can cause the model to miss relevant relations between features and target outputs (underfitting). **Variance:** is the error due to excessive sensitivity to small fluctuations in the training set. High variance can cause the model be too flexible (overfitting).

7.3 Feature Engineering and Selection

Definition 22. **Feature Engineering** is the process of creating new features or modifying existing features to improve the performance of machine learning models. This can involve techniques such as normalization, scaling, encoding categorical variables, and creating interaction terms.

- Domain Knowledge: Use knowledge about the problem domain to create meaningful features.
- Data visualization: Explore the data to identify patterns and relationships that can inform feature creation.
- Systemtic transformations: Apply mathematical transformations (e.g., log, square root) to features to improve their distribution or relationships with the target variable.

Definition 23. **Feature Selection** is the process of selecting a subset of relevant features for use in model training. This can help to reduce overfitting, improve model interpretability, and decrease computational cost. Common feature selection techniques include:

- Filter Methods: Use statistical measures (e.g., correlation, mutual information) to rank features and select the top k features.
- Wrapper Methods: Use a predictive model to evaluate different subsets of features and select the subset that yields the best model performance (e.g., recursive feature elimination).
- Embedded Methods: Perform feature selection as part of the model training process (e.g., Lasso regression, decision tree feature importance).

8 Lecture 7: Principal Component Analysis (PCA)

8.1 Introduction to PCA

Definition 24. Principal Component Analysis (PCA) is an unsupervised learning used for dimensoinality reduction, noise filtering, data visualization, and feature selection.
The main ides of PCA is to find a new set coordinate system. THe new axis (principal components) are orthogonal to each other and the variance of data is maximized along the first few axis.

8.1.1 Problem formulation

Let $X_c \in \mathbb{R}^{N \times d}$ be the centered data matrix (mean of each feature is subtracted from the data). Let $\vec{x}^{(i)} \in \mathbb{R}^d$ be the i^{th} data point (row of X_c). PCA finds a direction (unit vector) $\vec{v_1} \in \mathbb{R}^d$ such that the variance of the projected data onto $\vec{v_1}$ is maximized. The projection of a data point $\vec{x}^{(i)}$ onto $\vec{v_1}$ is given by:

$$z^{(1)_i} = \vec{x}^{(i)^T} \vec{v_1}$$

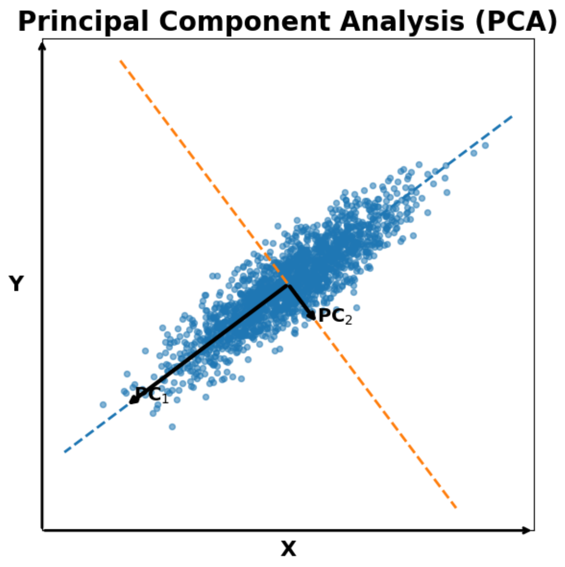


Figure 13: Principal Component Analysis: Finding new orthogonal axes that maximize variance

Collecting all the projected data points, we have:

$$z^{(1)} = \begin{pmatrix} z^{(1)_1} \\ z^{(1)_2} \\ \vdots \\ z^{(1)_N} \end{pmatrix} = X_c \vec{v_1}$$

$$\frac{1}{N}\sum_{i=1}^{N} z^{(1)_i} = \frac{1}{N}\sum_{i=1}^{N} \vec{x}^{(i)^T} \vec{v_1} = \vec{v_1}^T \left(\frac{1}{N}\sum_{i=1}^{N} \vec{x}^{(i)} \right) = 0$$

The Sample variance:

$$\begin{aligned}
\mathrm{Var}(z^{(1)}) &= \frac{1}{N-1}\sum_{i=1}^{N}(z^{(1)_i})^2 \\
&= \frac{1}{N-1}\sum_{i=1}^{N}(\vec{x}^{(i)^T}\vec{v_1})^2 \\
&= \frac{1}{N-1}\|X_c\vec{v_1}\|^2 \\
&= \frac{1}{N-1}\vec{v_1}^T X_c^T X_c \vec{v_1}
\end{aligned}$$

Now we let,

$$C = \frac{1}{N-1}X_c^T X_c$$

be the sample covariance matrix. So we have,

$$\mathrm{Var}(z^{(1)}) = \vec{v_1}^T C\vec{v_1} \quad \text{subject to} \quad \|\vec{v_1}\| = 1$$

In 1D, we would have that,

$$X(\mu, \sigma^2) \to aX(a\mu, a^2\sigma^2)$$

So scaling the data scales the variance. To avoid this, we constrain $\|\vec{v_1}\| = 1$.

8.2 Optimization Problem

Which unit vector $\vec{v_1}$ maximizes $\vec{v_1}^T C\vec{v_1}$?
We can use Lagrange multipliers to solve this constrained optimization problem. We define the Lagrangian function:

$$\mathcal{L}(\vec{v_1}, \lambda_1) = \vec{v_1}^T C\vec{v_1} - \lambda_1(\vec{v_1}^T\vec{v_1} - 1)$$

To find the stationary points, we take the gradient of the Lagrangian with respect to $\vec{v_1}$ and set it to zero:

$$\nabla_{\vec{v_1}}\mathcal{L} = 2C\vec{v_1} - 2\lambda_1\vec{v_1} = 0$$

This leads to the equation:

$$C\vec{v_1} = \lambda_1\vec{v_1} \implies \lambda_1 = \vec{v_1}^T C\vec{v_1}$$

The λ_1 is the variance in the direction of $\vec{v_1}$.
Now the subsequent principal components $\vec{v_2}, \vec{v_3}, \ldots, \vec{v_k}$ can be found by solving the same optimization problem with the additional constraint that each new principal component is orthogonal to all previously found components. For example, to find the second principal component $\vec{v_2}$, we solve:

$$\mathcal{L}(\vec{v_2}, \lambda_2) = \vec{v_2}^T C \vec{v_2} - \lambda_2(\vec{v_2}^T \vec{v_2} - 1) - \mu(\vec{v_2}^T \vec{v_1})$$

We require that $\vec{v_2}$ is orthogonal to $\vec{v_1}$, i.e., $\vec{v_2}^T \vec{v_1} = 0$. So,

$$\nabla_{\vec{v_2}} \mathcal{L} = 2C\vec{v_2} - 2\lambda_2 \vec{v_2} - \mu \vec{v_1} = 0$$

This leads to the equation:

$$C\vec{v_2} = \lambda_2 \vec{v_2} + \frac{\mu}{2} \vec{v_1}$$

We multiple both sides by $\vec{v_1}^T$ and use the orthogonality condition to find that $\mu = 0$. Thus, we have:

$$C\vec{v_2} = \lambda_2 \vec{v_2}$$

Continuing this process, we find that each principal component $\vec{v_k}$ satisfies the eigenvalue equation:

$$C\vec{v_k} = \lambda_k \vec{v_k} \quad \text{such that} \quad \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d \geq 0$$

Where λ_k is the variance along the direction of $\vec{v_k}$.

$$\sum_{k=1}^{d} \lambda_k = \text{trace}(C) = \sum_{j=1}^{d} C_{jj}$$

8.2.1 Example 1: Gene Expression Data

Toy dataset on gene expression:

Sample	Gene 1	Gene 2
Cell 1	2	3
Cell 2	3	3
Cell 3	8	7
Cell 4	7	8
Cell 5	1	2

Mean of each gene:

$$\bar{x_{G1}} = \frac{2+3+8+7+1}{5} = 4.2$$

$$\bar{x_{G2}} = \frac{3+3+7+8+2}{5} = 4.6$$

Center the data by subtracting the mean of each gene:

Sample	Gene 1 (centered)	Gene 2 (centered)
Cell 1	-2.2	-1.6
Cell 2	-1.2	-1.6
Cell 3	3.8	2.4
Cell 4	2.8	3.4
Cell 5	-3.2	-2.6

The covariance matrix is:

$$C = \begin{pmatrix} 9.7 & 8.1 \\ 8.1 & 7.3 \end{pmatrix}$$

To find the principal components, we solve the eigenvalue equation:

$$C\vec{v} = \lambda\vec{v} \implies (C - \lambda I)\vec{v} = 0 \implies \lambda_1 \approx 16.7, \lambda_2 \approx 0.3$$

The corresponding eigenvectors are:

$$\vec{v_1} \approx \begin{pmatrix} 0.757 \\ 0.653 \end{pmatrix}, \quad \vec{v_2} \approx \begin{pmatrix} -0.653 \\ 0.757 \end{pmatrix}$$

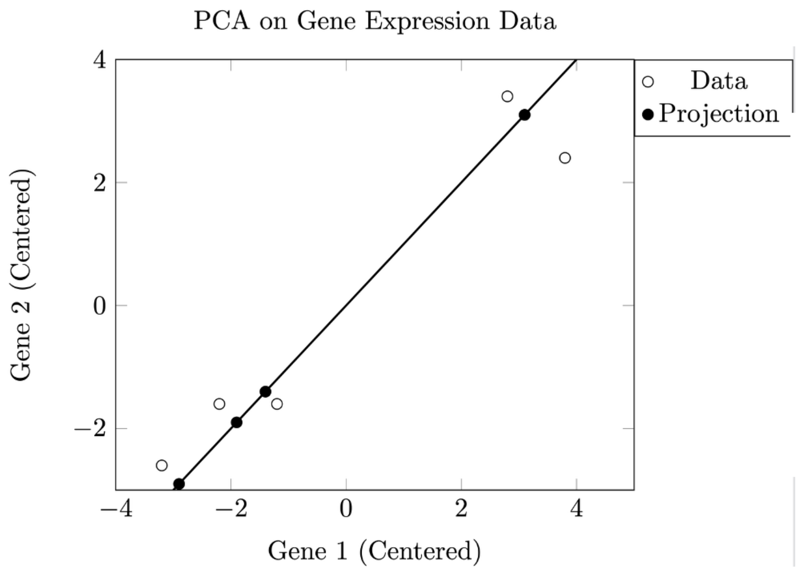


Figure 14: PCA Example: Gene Expression Data

Review 2. **Singular Value Decomposition (SVD)**: Any matrix $A \in \mathbb{R}^{m \times n}$ can be decomposed as:

$$A = U \Sigma V^T$$

Where:

- $U \in \mathbb{R}^{m \times m}$ is an orthogonal matrix whose columns are the left singular vectors of A.
- $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix with non-negative real numbers on the diagonal (singular values of A) arranged in descending order ($\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$, where r is the rank of A).
- $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix whose columns are the right singular vectors of A.

Now recall that,

$$C = \frac{1}{N-1} X_c^T X_c$$

So if we perform SVD on X_c:

$$X_c = U \Sigma V^T$$

Then,
$$C = \frac{1}{N-1} V\Sigma^T U^T U \Sigma V^T = \frac{1}{N-1} V \Sigma^2 V^T$$
Now we let,
$$\Lambda = \frac{1}{N-1}\Sigma^2$$
So,
$$C = V\Lambda V^T$$
Now the right singular vectors of X_c (columns of V) are the eigenvectors of C (principal components), and the eigenvalues of C are given by the squared singular values of X_c divided by $N-1$.
$$\lambda_k = \frac{\sigma_k^2}{N-1}$$

8.2.2 Example 2: User movie ratings

Consider the following toy dataset user-movie rating matrix:

User	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5
User 1	1	1	1	0	0
User 2	3	3	3	0	0
User 3	4	4	4	0	0
User 4	5	5	5	0	0
User 5	0	2	0	4	4
User 6	0	0	0	5	5
User 7	0	1	0	2	2

The rankings are from 0 (not watched) to 5 (liked it a lot).
The first 3 movies are sci-fi movies and the last 2 are romance movies.

Now let's use SVD:
$$U \approx \begin{pmatrix} 0.13 & 0.02 & -0.01 \\ 0.41 & 0.07 & -0.03 \\ 0.55 & 0.09 & -0.04 \\ 0.68 & 0.11 & -0.05 \\ 0.15 & -0.59 & 0.65 \\ 0.07 & -0.73 & -0.67 \\ 0.07 & -0.29 & 0.32 \end{pmatrix}, \quad \Sigma \approx \begin{pmatrix} 12.37 & 0 & 0 \\ 0 & 9.5 & 0 \\ 0 & 0 & 1.3 \end{pmatrix},$$
$$V^T \approx \begin{pmatrix} 0.56 & 0.59 & 0.56 & 0.09 & 0.09 \\ 0.12 & 0.02 & 0.12 & -0.69 & -0.69 \\ 0.4 & -0.8 & 0.4 & 0.9 & 0.9 \end{pmatrix}$$

Now notice that for U, the first column corresponds to user preference for sci-fi movies, and the second column corresponds to user preference for romance movies. The third column seems to capture some noise or less significant pattern in the data.

Similarly, for V^T, the first row corresponds to the sci-fi genre, and the second row corresponds to the romance genre. The third row again seems to capture some noise or less significant pattern in the data.

For Σ, the singular values indicate that the first two components (sci-fi and romance preferences) capture most of the variance in the data, while the third component contributes very little.

So we can approximate the original rating matrix using only the first two singular values and their corresponding singular vectors:

$$U_{reduced} = \begin{pmatrix} 0.13 & 0.02 \\ 0.41 & 0.07 \\ 0.55 & 0.09 \\ 0.68 & 0.11 \\ 0.15 & -0.59 \\ 0.07 & -0.73 \\ 0.07 & -0.29 \end{pmatrix}, \quad \Sigma_{reduced} = \begin{pmatrix} 12.37 & 0 \\ 0 & 9.5 \end{pmatrix},$$

$$V^T_{reduced} = \begin{pmatrix} 0.56 & 0.59 & 0.56 & 0.09 & 0.09 \\ 0.12 & 0.02 & 0.12 & -0.69 & -0.69 \end{pmatrix}$$

The approximated rating matrix is given by:

$$R_{approx} = U_{reduced}\Sigma_{reduced}V^T_{reduced}$$

This low-rank approximation captures the main patterns in user preferences while reducing noise and less significant variations in the data.
This shows how PCA (via SVD) can be used for dimensionality reduction and feature extraction in recommendation systems.

9 Lecture 8: Fisher's Linear Discriminant Analysis (FDA)

9.1 Introduction to FDA

Definition 25. Fisher's Linear Discriminant Analysis (FDA) is a supervised learning technique used for dimensionality reduction and classification. The main idea of FDA is to find a linear combination of features that maximizes the separation between different classes while minimizing the variance within each class. Making it easier to classify data points in a lower-dimensional space.
Key idea: Finding a direction $\vec{w}$ in the feature space such that

- The means of two classes are as far apart as possible when projected onto $\vec{w}$ (good class separability).
- The data within each class is as close together as possible when projected onto $\vec{w}$ (minimized within-class variance).

9.2 Betweeen class scatter matrix S_B

Suppose we have,

- x_i: feature space for sample i.
- $y_i \in \{0, 1\}$: class label for sample i.
- n_i: number of samples in class i.

The between-class scatter matrix measures the separation between the class means. It is defined as:

$$S_B = n_0(\vec{m}_0 - \vec{m})(\vec{m}_0 - \vec{m})^T + n_1(\vec{m}_1 - \vec{m})(\vec{m}_1 - \vec{m})^T$$

Where:

- $\vec{m}_0$ and $\vec{m}_1$ are the mean vectors of class 0 and class 1, respectively.
- $\vec{m}$ is the overall mean vector of the entire dataset.
- n_0 and n_1 are the number of samples in class 0 and class 1, respectively.

doing some algebra, we can rewrite S_B as:

$$\begin{aligned} S_B = n_0 \left(\vec{m}_0 - \frac{n_0\vec{m}_0 + n_1\vec{m}_1}{n_0+n_1}\right)\left(\vec{m}_0 - \frac{n_0\vec{m}_0 + n_1\vec{m}_1}{n_0+n_1}\right)^T \\ + n_1 \left(\vec{m}_1 - \frac{n_0\vec{m}_0 + n_1\vec{m}_1}{n_0+n_1}\right)\left(\vec{m}_1 - \frac{n_0\vec{m}_0 + n_1\vec{m}_1}{n_0+n_1}\right)^T \end{aligned}$$

$$\begin{aligned} \vec{m}_0 - \vec{m} &= \frac{n_1}{n_0+n_1}(\vec{m}_0 - \vec{m}_1) \\ \vec{m}_1 - \vec{m} &= \frac{n_0}{n_0+n_1}(\vec{m}_1 - \vec{m}_0) \end{aligned}$$

$$\implies S_B = \frac{n_0 n_1}{n_0+n_1}(\vec{m}_0 - \vec{m}_1)(\vec{m}_0 - \vec{m}_1)^T$$

Now, the direction that maximizes the between-class scatter is given by:

$$\begin{aligned} (w^T\vec{m}_1 - w^T\vec{m}_0)^2 &= w^T(\vec{m}_1 - \vec{m}_0)(\vec{m}_1 - \vec{m}_0)^T w \\ &= \frac{n_0+n_1}{n_0 n_1} w^T S_B w \end{aligned}$$

This implies that maximizing the between-class scatter is equivalent to maximizing $w^T S_B w$.

$$(w^T\vec{m}_1 - w^T\vec{m}_0)^2 \propto w^T S_B w$$

9.3 Within class scatter matrix S_W

The within-class scatter matrix measures the spread of data points within each class. It is defined as:

$$S_W = \sum_{i:y_i=0} (\vec{x_i} - \vec{m}_0)(\vec{x_i} - \vec{m}_0)^T + \sum_{i:y_i=1} (\vec{x_i} - \vec{m}_1)(\vec{x_i} - \vec{m}_1)^T$$

Where:

- $\vec{x_i}$ is the feature vector of sample i.
- $\vec{m}_0$ and $\vec{m}_1$ are the mean vectors of class 0 and class 1, respectively.
- y_i is the class label for sample i.

The direction that minimizes the within-class scatter is given by:

$$\sum_{i:y_i=0}(w^T\vec{x_i} - w^T\vec{m_0})^2 + \sum_{i:y_i=1}(w^T\vec{x_i} - w^T\vec{m_1})^2$$
$$= w^T S_W w$$

This implies that minimizing the within-class scatter is equivalent to minimizing $w^T S_W w$.

9.4 FDA Optimization Problem

The goal of FDA is to find the direction $\vec{w}$ that maximizes $w^T S_B w$ while minimizing $w^T S_W w$. This can be formulated as the following optimization problem:
The first approach is the to maximize the ratio of between-class scatter to within-class scatter:

$$\max_w J(w) = \frac{w^T S_B w}{w^T S_W w} = \frac{N(w)}{D(w)}$$

Now the gradient of $J(w)$ is given by:

$$\nabla_w J(w) = \frac{D(w)\nabla N(w) - N(w)\nabla D(w)}{D(w)^2} = 0$$

Where:

$$\nabla N(w) = 2S_B w, \quad \nabla D(w) = 2S_W w$$

Now the alternate approach is to maximize the numerator while keeping the denominator constant (i.e., $w^T S_W w = 1$):

$$\max_w w^T S_B w \quad \text{subject to} \quad w^T S_W w = 1$$

We can use Lagrange multipliers to solve this constrained optimization problem. We define the Lagrangian function:

$$\mathcal{L}(w, \lambda) = w^T S_B w - \lambda(w^T S_W w - 1)$$

To find the stationary points, we take the gradient of the Lagrangian with respect to w and set it to zero:

$$\nabla_w \mathcal{L} = 2S_B w - 2\lambda S_W w = 0$$

This leads to the equation:

$$S_B w = \lambda S_W w \implies S_W^{-1} S_B w = \lambda w$$

This is a generalized eigenvalue problem. The optimal direction w is given by the eigenvector corresponding to the largest eigenvalue λ of the matrix $S_W^{-1} S_B$.

9.4.1 Special Case: Two Classes

In the case of two classes, the between-class scatter matrix S_B is of rank 1. To see this, we can express S_B in terms of the difference between the class means: Let,

$$\vec{d} = \sqrt{\frac{n_0 n_1}{n_0 + n_1}}(\vec{m}_1 - \vec{m}_0)$$

So,

$$S_B = \frac{n_0 n_1}{n_0 + n_1}(\vec{m}_1 - \vec{m}_0)(\vec{m}_1 - \vec{m}_0)^T = \vec{d}\vec{d}^T$$

Thus,

$$S_W^{-1} S_B w = S_W^{-1} \vec{d}\vec{d}^T w = \lambda w$$

This implies that w is in the direction of $S_W^{-1}\vec{d}$. Therefore, the optimal direction w can be computed as:

$$w \propto S_W^{-1}(\vec{m}_1 - \vec{m}_0)$$

Optionally, we can have $normw = 1$.

9.5 Example: Bacterial vs. Viral Infections

Consider a dataset with two classes: bacterial infections (class 0) and viral infections (class 1). Each sample has two features: CRP (C-reactive protein level in mg/L) and temperature (in °C). The dataset is as follows:

Viral		**Bacterial**	
CRP	**T**	**CRP**	**T**
90	36.0	42.0	37.6
11.1	37.2	31.1	42.2
30.0	36.5	50.0	38.5
21.4	39.4	60.4	39.4
10.7	39.6	43.7	38.6
3.4	40.7	17.3	42.7

Table 1: CRP levels and temperature for viral and bacterial cases

Now the ranges for CPR is (0-100) mg/L and temperature is (35-43) °C.

- 0-5 mg/L: normal
- 5-20 mg/L: Viral infection

- 20-100 mg/L: Bacterial or Viral infection
- >100 mg/L: bacterial infection

We let clases 0 be the viral infections and class 1 be the bacterial infections. We have $n_0 = 6$ and $n_1 = 6$.
Now,

$$\vec{x} = \begin{pmatrix} \text{CRP} \\ \text{Temperature} \end{pmatrix}$$

So Patient 1 with viral infection has,

$$\vec{x}^{(1)} = \begin{pmatrix} 90 \\ 36.0 \end{pmatrix}$$

The mean vectors for each class are:

$$\vec{m}_0 = \frac{1}{n_0} \sum_{i=0} \vec{x}_{(i)} = \begin{pmatrix} 19.4 \\ 38.2 \end{pmatrix}, \quad \vec{m}_1 = \frac{1}{n_1} \sum_{i=1} \vec{x}_{(i)} = \begin{pmatrix} 41.1 \\ 39.8 \end{pmatrix}$$

The within-class scatter matrix is:

$$S_K = \sum_{i:y_i=0} (\vec{x_i} - \vec{m_0})(\vec{x_i} - \vec{m_0})^T + \sum_{i:y_i=1} (\vec{x_i} - \vec{m_1})(\vec{x_i} - \vec{m_1})^T$$

Now,

$$S_W = S_0 + S_1 = \begin{pmatrix} 2081.4 & -224.6 \\ -224.6 & 40.7 \end{pmatrix}$$

Now,

$$\vec{w} \propto S_W^{-1}(\vec{m}_1 - \vec{m}_0) = \begin{pmatrix} 0.03 \\ 0.24 \end{pmatrix}$$

Normalizing $\vec{w}$:

$$\vec{w} = \frac{1}{\|\vec{w}\|} \vec{w} = \begin{pmatrix} 0.1496 \\ 0.9887 \end{pmatrix}$$

The projection of a data point $\vec{x}$ onto the direction $\vec{w}$ is given by:

$$z = \vec{w}^T \vec{x} = 0.1496 \cdot \text{CRP} + 0.9887 \cdot \text{Temperature}$$

Now,

$$z_0 = \vec{w}^T \vec{m}_0 \approx 40.7$$

and,

$$z_1 = \vec{w}^T \vec{m}_1 \approx 45.5$$

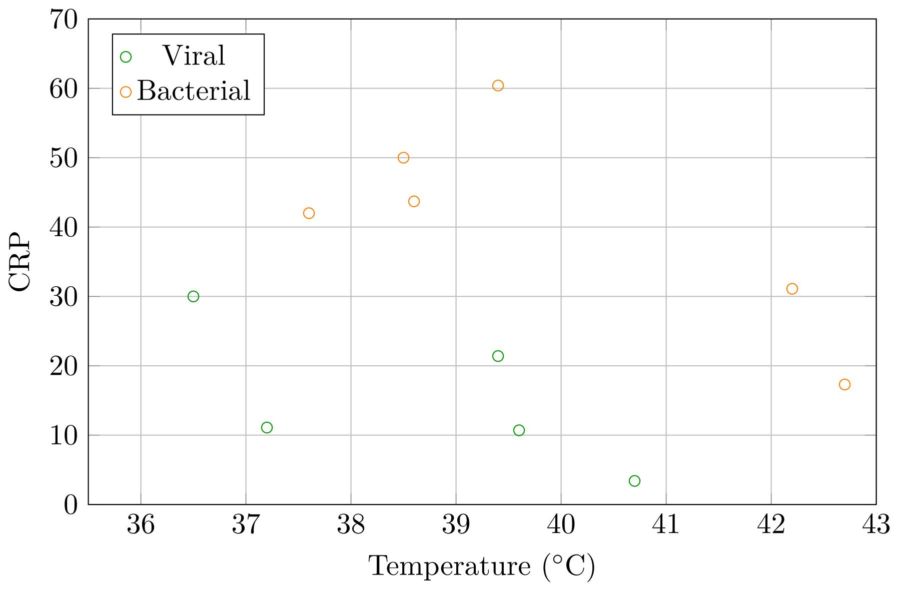


Figure 15: CRP versus Temperature for Viral and Bacterial Infections

A reasonable threshold to classify new patients could be the midpoint between z_0 and z_1:

$$z_{threshold} = \frac{z_0 + z_1}{2} \approx 43.1$$

So if a new patient's projected value z is less than 43.1, we classify them as having a viral infection (class 0). If z is greater than or equal to 43.1, we classify them as having a bacterial infection (class 1).

9.6 Using FDA for feature selection

FDA can also be used for feature selection by analyzing the weights in the projection vector w. Features with higher absolute weights contribute more to class separability and are considered more important. By selecting features with the highest weights, we can reduce the dimensionality of the dataset while retaining the most discriminative information for classification tasks.
So,

$$z = w^T x = \sum_{i=1}^{d} w_i x_i$$

Where w_i is the weight corresponding to feature x_i. Features with larger $|w_i|$ values are more important for distinguishing between classes.

10 Lecture 9: Clustering

10.1 Introduction to Clustering

Definition 26. **Clustering** is an unsupervised learning technique used to group similar data points together based on their features. The goal of clustering is to identify natural groupings or patterns in the data without any prior knowledge of class labels. Types of clustering:

- Hard Clustering: Each data point belongs to exactly one cluster (e.g., K-means clustering).
- Soft Clustering: Each data point can belong to multiple clusters with varying degrees of membership (e.g., Gaussian Mixture Models).

Warning

Clustering is model/algorithm dependent (no single correct clustering). Different algorithms may produce different clusterings on the same dataset. It is important to choose an appropriate clustering algorithm based on the characteristics of the data and the specific application.

10.2 K-means Clustering

Definition 27. K-means clustering is a popular hard clustering algorithm that partitions the data into K clusters by dividing points into K groups such that the each point is assigned to the cluster with the nearest mean (centroid). The algorithm iteratively refines the cluster assignments and centroids until simeqergence.

10.2.1 Mathematical formulation

Let,

$$x_i \in \mathbb{R}^d : \text{data point } i$$
$$\mu_j \in \mathbb{R}^d : \text{centroid of cluster } j$$
$$k : \text{ is the pre-defined number of clusters}$$

The objective of K-means is to minimize the within-cluster sum of squares (WCSS):

$$\min_{\{\mu_j\}_{j=1}^k} \sum_{i=1}^{N} \min_{1\le j\le k} \|x_i - \mu_j\|^2$$

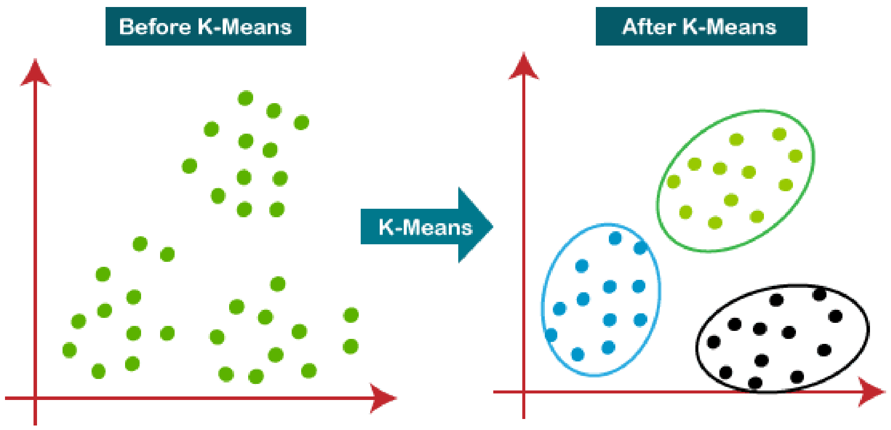


Figure 16: K-means Clustering: Iteratively refining cluster assignments and centroids

10.2.2 K-means Algorithm

The K-means algorithm consists of the following steps:

```
Input: Data points $X = \{x_1, x_2, \ldots, x_N\}$, number of clusters $K$
Output: Cluster assignments for each data point, final cluster
        centroids
Initialization: Randomly select $K$ data points as initial centroids
  $\{\mu_1, \mu_2, \ldots, \mu_K\}$;
while not simeqerged do
    Assignment Step: Assign each data point to the nearest
      centroid:
        $$C_i = \arg\min_{1 \le j \le K} \|x_i - \mu_j\|^2 \quad \forall i = 1, 2, \ldots, N$$
    ;
    Update Step: Update the centroids by calculating the mean of
      the assigned data points:
        $$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i \quad \forall j = 1, 2, \ldots, K$$
    ;
end
```

Basically the algorithm does the following:

- Randomly initialize K centroids.
- Assign each data point to the nearest centroid to form K clusters.
- Update the centroids by calculating the mean of the data points in each cluster.
- Repeat the assignment and update steps until the centroids do not change significantly (simeqergence).

Now there are limitations to K-means clustering:

- K-means assumes that clusters are spherical and equally sized, which may not always be the case in real-world data.
- The algorithm is sensitive to the initial placement of centroids, which can lead to different results on different runs (local minima).

- K-means is not suitable for clustering non-simeqex shapes or clusters with varying densities.
- The number of clusters K must be specified in advance, which can be challenging to determine.

There are various methods to address these limitations, such as using K-means++ for better initialization. K-means++ initializes centroids in a way that spreads them out, which can lead to better simeqergence and improved clustering results.

```
**Input:** Data points $X = \{x_1, x_2, \ldots, x_N\}$, number of clusters $K$
**Output:** Initial centroids for K-means
**Initialization:** Randomly select the first centroid $\mu_1$ from the data
  points;
for $j$ = 2 to $K$ do
    Compute the distance $D(x_i)$ from each data point $x_i$ to the
      nearest existing centroid:
        $D(x_i) = \min_{1 \leq l < j} \|x_i - \mu_l\|^2 \quad \forall i = 1, 2, \ldots, N$
      ;
    Select the next centroid $\mu_j$ from the data points with
      probability proportional to $D(x_i)^2$:
        $P(x_i) = \frac{D(x_i)^2}{\sum_{i=1}^{N} D(x_i)^2} \quad \forall i = 1, 2, \ldots, N$
      ;
end
```

There is also the Elbow Method to determine the optimal number of clusters K. The Elbow Method involves plotting the WCSS against different values of K and looking for an "elbow" point where the rate of decrease in WCSS slows down significantly. This point indicates a good trade-off between the number of clusters and the compactness of the clusters.

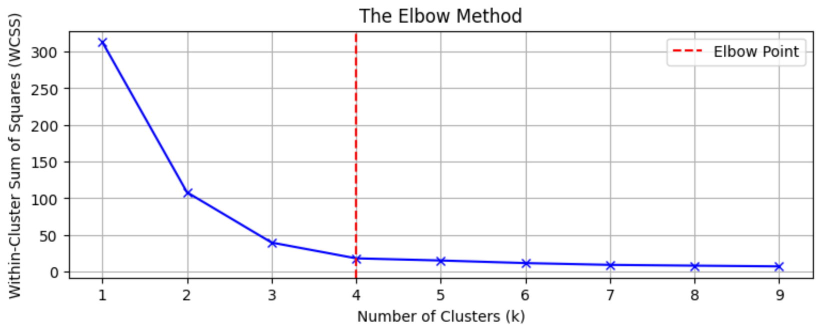


Figure 17: Elbow Method: Determining the optimal number of clusters K

10.3 Perceptron

Definition 28. The Perceptron is a supervised learning algorithm used for binary classification tasks. It is a type of linear classifier that maps input features to a binary output (0 or 1) using a linear decision boundary. The Perceptron algorithm iteratively adjusts the weights associated with the input features based on the classification errors made during training.
Key idea: Finding a hyperplane that separates the two classes in the feature space.

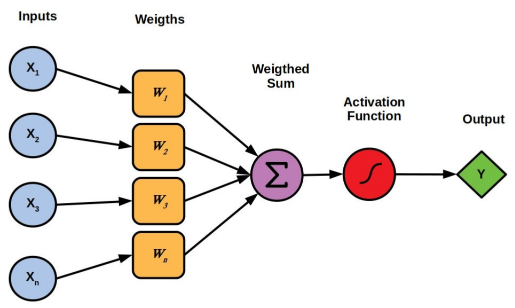


Figure 18: Perceptron: Finding a linear decision boundary to separate two classes

Here, $y = f(\vec{w}^T\vec{x} + b)$, where f is the step function:

$$f(z) = \begin{cases} +1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$$

10.3.1 Geometric Interpretation

The decision boundary of the Perceptron is defined by the equation:

$$\vec{w}^T\vec{x} + b = 0$$

This equation represents a hyperplane in the feature space that separates the two classes. The vector $\vec{w}$ is perpendicular to the hyperplane and determines its orientation, while the bias term b shifts the hyperplane away from the origin. The distance from a point $\vec{x}$ to the decision boundary can be calculated as:

$$\text{Distance} = \frac{\vec{w}^T\vec{x} + b}{\|\vec{w}\|}$$

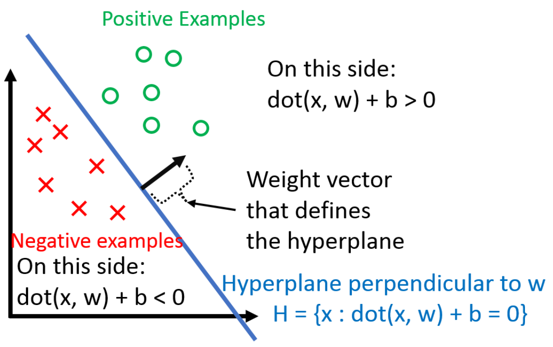


Figure 19: Geometric Interpretation of the Perceptron Decision Boundary

10.3.2 Minimizing Classification Error

The Perceptron algorithm aims to minimize the classification error by adjusting the weights $\vec{w}$ and bias b based on the misclassified data points. The classification error can be defined as:

$$Err(w, b) = -\sum_{i \in M} y_i(\vec{w}^T\vec{x_i} + b) \quad \frac{\partial Err}{\partial w} = -\sum_{i \in M} y_i\vec{x_i}, \quad \frac{\partial Err}{\partial b} = -\sum_{i \in M} y_i$$

Where M is the set of misclassified points. The weights and bias are updated as follows:

$$\vec{w}^{(new)} \leftarrow \vec{w}^{(old)} + \eta \sum_{i \in M} y_i \vec{x_i}$$

$$b^{(new)} \leftarrow b^{(old)} + \eta \sum_{i \in M} y_i$$

Where η is the learning rate. The algorithm iteratively updates the weights and bias until all data points are correctly classified or a maximum number of iterations is reached.

10.3.3 Multi-layer Perceptron (MLP)

A Multi-layer Perceptron (MLP) is a type of feedforward artificial neural network that consists of multiple layers of nodes (neurons) connected in a directed graph. MLPs are capable of learning complex non-linear mappings between input features and output labels, making them suitable for a wide range of tasks, including classification and regression.
An MLP typically consists of the following components:

- Input Layer: The first layer that receives the input features.
- Hidden Layers: One or more layers of neurons that process the input data and learn intermediate representations. Each neuron in a hidden layer applies a weighted sum of its inputs followed by a non-linear activation function (e.g., ReLU, sigmoid, tanh).

$$\sum_l = w^{(l)} x^{(l-1)} + b^{(l)}, \quad z_{(l)} = f(\sum_l)$$

- Output Layer: The final layer that produces the output predictions. The number of neurons in the output layer depends on the task (e.g., one neuron for binary classification, multiple neurons for multi-class classification).
- Weights and Biases: Each connection between neurons has an associated weight, and each neuron has a bias term. These parameters are learned during the training process.

Definition 29. **Depth:** The number of layers in the network (including input, hidden, and output layers). **Width:** The number of neurons in each layer.

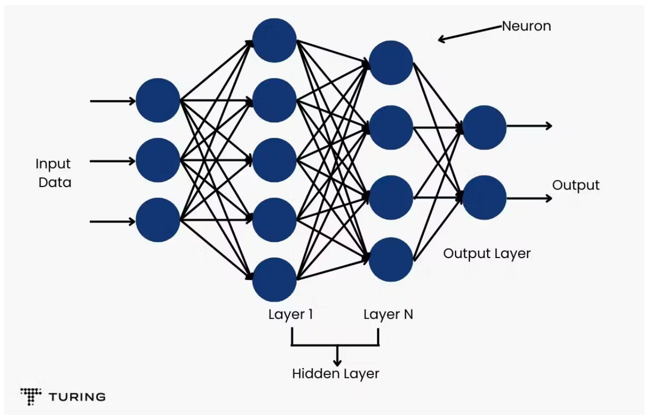


Figure 20: Multi-layer Perceptron (MLP): A feedforward neural network with input, hidden, and output layers

Note. Depth is generally more effective than width for learning complex functions. Deeper networks can introduce more non-linearities and hierarchical feature representations, allowing them to capture intricate patterns in the data.

10.3.4 Activation functions

Activation functions introduce non-linearity into the MLP, allowing it to learn complex patterns in the data. Common activation functions include:

- Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$
- Tanh: $f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLU (Rectified Linear Unit): $f(z) = \max(0, z)$
- Softmax: $f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$ (used for multi-class classification)

10.3.5 Universal Approximation Theorem

Theorem 1. The Universal Approximation Theorem states that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$, given appropriate activation functions (non-constant, bounded and continuous, e.g., sigmoid, ReLU).

Property 6.

- Exisience is guaranteed, but the network size is not specified.
- Deep networks (multiple hidden layers) can achieve similar approximation accuracy with fewer neurons compared to shallow networks (single hidden layer).
- The theorem does not provide a method for finding the optimal weights and biases for the network.

10.3.6 Training MLPs

Training MLPs involves minimizing a loss function that measures the difference between the predicted outputs and the true labels. Common loss functions include:

- Mean Squared Error (MSE): Used for regression tasks.

  $$L(w, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

  Where y_i is the true label and $\hat{y}_i$ is the predicted output.

- MAE (Mean Absolute Error): Also used for regression tasks.

  $$L(w, b) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

- Cross-Entropy Loss: Used for classification tasks.

  $$L(w, b) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

11 Lecture 10: Deep Neural Networks and Training

11.1 Deep Neural Networks

Definition 30. **Deep Neural Network (DNN)** is a neural network with multiple hidden layers between the input and output layers, allowing it to learn complex and abstract representations of the input data. The architecture consists of:

- **Input Layer:** Receives the input features
- **Hidden Layers:** Multiple layers that process and transform the data
- **Output Layer:** Produces the final predictions

Definition 31. **Width** of a DNN refers to the number of neurons in a layer.
Depth of a DNN refers to the number of layers in the network.

Note. Enhancing the width and depth of a Neural Network allows for the representation of increasingly complex input/output relationships. Generally, depth is more effective than width for learning complex functions, as deeper networks can introduce more non-linearities and hierarchical feature representations.

11.2 Activation Functions

Definition 32. **Activation functions** introduce non-linearity into neural networks. Without them, no matter how many layers we stack, the network would behave like a single linear model.

11.2.1 Common Activation Functions

1. Sigmoid (Logistic) Activation Function

The sigmoid function maps any real number to a value between 0 and 1:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Properties:

- Output range: $(0, 1)$
- Suitable for predicting probabilities in binary classification
- **Drawback:** Suffers from vanishing gradient problem during back-propagation
- When inputs are very large or very small, the gradient becomes close to zero, which slows learning in deep networks

2. Hyperbolic Tangent (tanh) Function
The tanh function is a scaled version of the sigmoid:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Properties:

- Output range: $(-1, 1)$
- Zero-centered, making it easier for the model to learn in some cases
- Positive and negative activations are balanced, which often helps optimization
- **Drawback:** Still suffers from vanishing gradients for large inputs

3. Rectified Linear Unit (ReLU)
ReLU is one of the most widely used activation functions:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Properties:

- Allows only positive values to pass through
- Computationally efficient due to its linear, non-saturating form
- Avoids saturation in the positive region, helping mitigate vanishing gradients
- Particularly effective in simeqolutional neural networks
- **Limitation:** "Dying ReLU" problem - neurons can get stuck outputting zero if their weights push them into the negative region permanently

Note. Variants like Leaky ReLU are sometimes used to address the dying ReLU problem.

4. Softmax Function
The softmax function is used for multi-class classification:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Properties:

- simeqerts raw scores (logits) into probabilities that sum to 1
- Each output lies between 0 and 1
- Used primarily in the output layer for multi-class classification tasks
- The highest value corresponds to the predicted class
- **Important:** Not used in hidden layers; specifically designed for final predictions

11.3 Loss Functions

11.3.1 Common Loss Functions for Neural Networks

1. Mean Squared Error (MSE)
Used for regression tasks:

$$L(w, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i is the true label and $\hat{y}_i$ is the predicted output.
2. Mean Absolute Error (MAE)
Also used for regression tasks:

$$L(w, b) = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

3. Cross-Entropy Loss
Used for classification tasks:

$$L(w, b) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

11.4 Backpropagation

Definition 33. **Backpropagation**, short for "backward propagation of errors," is a widely used algorithm in training feedforward neural networks for supervised learning tasks. It efficiently computes the gradients of the loss function with respect to the network parameters using the chain rule.

11.4.1 Network Architecture

Consider a simple network consisting of:

- Input layer
- One hidden layer
- Output layer

11.4.2 Forward Pass

Let:

$$\begin{aligned} x &: \text{input vector} \\ W^{(1)} &: \text{weights from input to hidden layer} \\ b^{(1)} &: \text{bias for hidden layer} \\ W^{(2)} &: \text{weights from hidden to output layer} \\ b^{(2)} &: \text{bias for output layer} \\ f &: \text{activation function} \end{aligned}$$

The forward pass computes:

$$\begin{aligned} z^{(1)} &= W^{(1)}x + b^{(1)} \\ a^{(1)} &= f(z^{(1)}) \\ z^{(2)} &= W^{(2)}a^{(1)} + b^{(2)} \\ \hat{y} &= f(z^{(2)}) \end{aligned}$$

11.4.3 Backward Pass with Least Squares Loss

For the least squares loss $L = \frac{1}{2}(y - \hat{y})^2$, the gradients are computed using the chain rule:

Output Layer Gradients:

$$\begin{aligned}\frac{\partial L}{\partial \hat{y}} &= -(y - \hat{y}) \\ \frac{\partial L}{\partial z^{(2)}} &= \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} = -(y - \hat{y}) \cdot f'(z^{(2)})\end{aligned}$$

Hidden Layer Gradients:

$$\begin{aligned}\frac{\partial L}{\partial a^{(1)}} &= \frac{\partial L}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial a^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \cdot W^{(2)} \\ \frac{\partial L}{\partial z^{(1)}} &= \frac{\partial L}{\partial a^{(1)}} \cdot \frac{\partial a^{(1)}}{\partial z^{(1)}} = \frac{\partial L}{\partial a^{(1)}} \cdot f'(z^{(1)})\end{aligned}$$

Weight and Bias Gradients:

$$\begin{aligned}\frac{\partial L}{\partial W^{(2)}} &= \frac{\partial L}{\partial z^{(2)}} \cdot (a^{(1)})^T \\ \frac{\partial L}{\partial b^{(2)}} &= \frac{\partial L}{\partial z^{(2)}} \\ \frac{\partial L}{\partial W^{(1)}} &= \frac{\partial L}{\partial z^{(1)}} \cdot x^T \\ \frac{\partial L}{\partial b^{(1)}} &= \frac{\partial L}{\partial z^{(1)}}\end{aligned}$$

Parameter Updates:
The weights and biases are updated using gradient descent:

$$\begin{aligned}W^{(2)} &\leftarrow W^{(2)} - \eta \frac{\partial L}{\partial W^{(2)}} \\ b^{(2)} &\leftarrow b^{(2)} - \eta \frac{\partial L}{\partial b^{(2)}} \\ W^{(1)} &\leftarrow W^{(1)} - \eta \frac{\partial L}{\partial W^{(1)}} \\ b^{(1)} &\leftarrow b^{(1)} - \eta \frac{\partial L}{\partial b^{(1)}}\end{aligned}$$

where η is the learning rate.

11.5 Optimization Challenges

11.5.1 Non-simeqex Loss Landscapes

Warning

Ideal loss functions are "nice bowls" (simeqex) with a single minimum, but neural network loss functions are non-simeqex due to complex, non-linear transformations from multiple layers and nonlinear activation functions.

Key Challenges:

- **Local Minima:** Points where the loss is lower than nearby points but not globally optimal. Gradient descent can get stuck in these local minima because the gradient becomes zero.
- **Saddle Points:** Locations where the gradient is zero, but the point is not a minimum - it is a minimum in one direction and a maximum in another. In high-dimensional neural networks, saddle points are often a bigger practical problem than local minima. Gradient descent can slow down or stall here because the gradient vanishes.
- **Plateaus:** Flat regions where gradients are very small, causing slow progress.

Note. Deep neural network loss surfaces are highly complex with many peaks, valleys, and flat plateaus, making optimization challenging.

11.6 Gradient Descent Variants

11.6.1 Gradient Descent vs Stochastic Gradient Descent

Gradient Descent (GD):

- Computes the gradient using the entire training dataset
- Update rule: $\theta \leftarrow \theta - \eta \nabla_\theta L(\theta)$
- More stable but computationally expensive for large datasets
- May simeqerge to local minima

Stochastic Gradient Descent (SGD):

- Computes the gradient using a single randomly selected training example

- Update rule: $\theta \leftarrow \theta - \eta \nabla_\theta L(\theta; x_i, y_i)$
- Much faster per iteration
- Updates are noisy, which can help escape local minima
- May not simeqerge to the exact minimum

11.6.2 Mini-Batch Gradient Descent

Definition 34. **Mini-Batch Gradient Descent** is a compromise between GD and SGD that uses a small batch of training examples to compute each gradient update.

Properties:

- Balances computational efficiency and stability
- Update rule: $\theta \leftarrow \theta - \eta \nabla_\theta L(\theta; \{x_i, y_i\}_{i \in \text{batch}})$
- Common batch sizes: 32, 64, 128, 256
- Can leverage vectorization and GPU parallelization
- Provides a good trade-off between simeqergence speed and stability

11.6.3 Momentum

Definition 35. **Momentum** is a technique that helps accelerate gradient descent by accumulating a velocity vector in directions of persistent reduction in the loss function.

The momentum update rule is:

$$v_t = \beta v_{t-1} + \eta \nabla_\theta L(\theta)$$
$$\theta \leftarrow \theta - v_t$$

where:

- v_t is the velocity at time t
- β is the momentum coefficient (typically 0.9)
- η is the learning rate

Benefits:

- Helps navigate ravines in the loss landscape
- Dampens oscillations
- Can help escape shallow local minima and saddle points
- Accelerates simeqergence in relevant directions

11.7 Learning Rate

Definition 36. The **learning rate** is a crucial hyperparameter in training neural networks, controlling the step size at each iteration while moving towards a minimum of the loss function.

Learning Rate Too Small:

- Slow simeqergence
- Risk of not simeqerging in reasonable time
- May get stuck in poor local minima
- Training becomes inefficient

Learning Rate Too Large:

- Overshooting the minimum
- Oscillations around the minimum
- Divergence - the loss may grow exponentially
- Training becomes unstable

Note. Test a variety of learning rates to find the one that works just right. Monitor the training loss to identify a value that leads to steady and reliable simeqergence.

11.7.1 Adaptive Learning Rate Methods

Definition 37. **Adaptive learning rate** techniques dynamically adjust the learning rate during a model's training. These methods are designed to improve the simeqergence of the model by ensuring that it neither overshoots nor gets stuck and is able to navigate the loss landscape more effectively.

The learning rate can adjust, becoming either larger or smaller, based on:

- The magnitude of the gradient
- The speed of learning
- The size of specific weights

Common Gradient Descent Algorithms (implemented in `torch.optim`):

- **SGD (Stochastic Gradient Descent):** Basic gradient descent with optional momentum
- **AdaGrad (Adaptive Gradient Algorithm):** Adapts learning rate based on historical gradients
- **RMSProp (Root Mean Square Propagation):** Uses moving average of squared gradients
- **Adam (Adaptive Moment Estimation):** Combines momentum and adaptive learning rates
- **AdaDelta:** Extension of AdaGrad that reduces aggressive learning rate decay

Note. Adaptive methods allow larger updates for infrequent parameters and smaller updates for frequently changing ones, leading to more stable and efficient training in practice.

11.8 Regularization Techniques

Regularization techniques help prevent overfitting and improve the generalization of neural networks to unseen data.

11.8.1 Dropout

Definition 38. **Dropout** is a regularization technique where randomly selected neurons are ignored (dropped out) during training. This prevents neurons from co-adapting too much and forces the network to learn more robust features.

How it works:

- During training, each neuron has a probability p (typically 0.5) of being temporarily removed
- Different neurons are dropped in each training iteration
- During testing, all neurons are used, but their outputs are scaled by the dropout probability
- Acts as an ensemble method by training many different network architectures

11.8.2 Early Stopping

Definition 39. **Early Stopping** is a regularization technique where the training process is stopped as soon as the performance on a held-out validation set stops improving.

Procedure:

- The model is evaluated on both the training set and a separate validation set during training
- The training loss typically keeps decreasing as the model continues to fit the data
- The validation loss initially decreases (model generalizing well) but eventually starts to increase when the model begins to overfit
- The optimal stopping point is when the validation loss reaches its minimum - just before it starts to rise

Note. Early stopping provides a simple yet effective way to prevent overfitting without modifying the network architecture or adding explicit regularization terms to the loss function.

11.8.3 Dataset Augmentation

Definition 40. **Data augmentation** is a technique used to artificially increase the size of a training dataset by generating modified versions of existing data. It is a form of regularization because it helps prevent the model from overfitting to the specific patterns or noise present in the training data, encouraging better generalization to unseen data.

In data augmentation, new data points are created by applying transformations to the original training examples. These transformations modify the original data without changing its meaning or label. This process increases the diversity of the training set, making the model more robust to variations in the input.

Example: For Image Data

- **Rotation:** Rotate images by small angles
- **Flipping:** Flip images horizontally or vertically
- **Scaling:** Resize images to different sizes
- **Cropping:** Randomly crop parts of images
- **Brightness/Contrast Adjustments:** Simulate different lighting conditions

Example: For Text Data

- **Synonym Replacement:** Replace words with synonyms
- **Back Translation:** Translate sentences to another language and back
- **Random Insertion/Deletion:** Add or remove words to modify context

Note. Data augmentation preserves the semantic meaning and label of the data while increasing variability, making the model more robust to variations in real-world data.

11.8.4 Noise Injection

Definition 41. **Noise injection** is a data augmentation technique that adds random noise to the original data to make the model more robust to variations, noise, or imperfections in real-world data. The goal is to help the model generalize better by training it on slightly altered versions of the original data, simulating the noise it might encounter in practice.

Example in Practice:

Consider an image classification task where the model is trained to identify objects like cats and dogs. In the real world, not all images will be perfect - some might be blurry, underexposed, or have digital artifacts. By injecting noise (like Gaussian noise) into the training images, the model learns to identify the object (e.g., a cat) even when the image quality is degraded or imperfect.

Benefits of Noise Injection:

- **Improved Generalization:** The model learns features that are less sensitive to small perturbations in the input
- **Prevents Overfitting:** Prevents the model from memorizing exact training examples
- **Enhances Robustness:** Improves performance on noisy or imperfect real-world data

11.8.5 Parameter Tying (or Sharing)

Definition 42. **Parameter tying** (or parameter sharing) refers to a technique used in machine learning models, particularly in deep learning, where the same set of parameters (weights) is reused across different parts of the model. Instead of learning a separate set of weights for each layer or part of the model, a single set of weights is "tied" or "shared" across multiple parts of the model.

This approach has several benefits, including:

- Reducing the number of parameters the model needs to learn
- Improving generalization
- Introducing invariance properties

Common Applications:

- **CNNs (simeqolutional Neural Networks):** Filters (kernels) are shared across different spatial locations of an image
- **RNNs (Recurrent Neural Networks):** The same weights are applied at each time step
- **Transformers:** Weights may be shared across layers for efficiency

11.8.6 Batch Normalization

Definition 43. **Batch normalization** is a technique used to improve the training of deep neural networks by normalizing the inputs to each layer within a mini-batch. It helps stabilize and accelerate training.

Introduced by Sergey Ioffe and Christian Szegedy in 2015, batch normalization addresses the issue of **internal covariate shift** - the phenomenon where the distribution of inputs to a neural network's layers changes during training, which can slow down the simeqergence of the model.

11.8.7 Internal Covariate Shift

Definition 44. **Internal covariate shift** refers to the phenomenon where the distribution of the inputs to each layer of a neural network changes during training as the network's parameters (weights and biases) are updated.

In a deep neural network, the inputs to each layer are typically influenced by the parameters (weights) of the previous layers. As these parameters are updated during training, the distribution of the inputs to each layer can change, causing instability and slowing down learning. Batch normalization works by standardizing the inputs to each layer, ensuring that they have a stable distribution, which helps in faster and more efficient learning.

11.8.8 How Batch Normalization Works

For each mini-batch of data passing through a layer, Batch Normalization performs two steps:

1. Standardize:

For a mini-batch $\mathcal{B} = \{x_1, x_2, \ldots, x_m\}$, compute:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{(batch mean)}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \quad \text{(batch variance)}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad \text{(normalize)}$$

where ϵ is a small constant for numerical stability.

2. **Re-scale and Re-shift:**

$$y_i = \gamma \hat{x}_i + \beta$$

where γ (scale) and β (shift) are learnable parameters. This gives the network the flexibility to undo the normalization if that's better for the task.

Note. Internal Covariate Shift was originally proposed as the main reason Batch Normalization improves training. However, more recent studies suggest that while reducing ICS is part of the story, it may not be the primary reason for Batch Normalization's effectiveness.

Batch Normalization works for multiple reasons, including:

- Smoothing the optimization landscape, making gradient-based optimization more stable and predictable
- Enabling higher learning rates, which can significantly speed up training
- Acting as a form of regularization by adding noise through mini-batch statistics
- Stabilizing gradients, making it easier to train very deep networks

11.9 Training Process

11.9.1 Epochs

Definition 45. An **epoch** refers to one complete pass through the entire training dataset. During an epoch, the model will have had the opportunity to learn from every example in the training set once.

During a typical epoch:

- **Forward Pass:** The model takes each training input and makes a prediction
- **Loss Computation:** Compare the model's prediction to the true label using a loss function (e.g., MSE for regression, cross-entropy for classification)
- **Backward Pass:** The gradients of the loss with respect to the model parameters are computed using backpropagation

- **Parameter Update:** An optimizer (e.g., Gradient Descent, SGD, Adam) adjusts the weights to reduce the loss

Once all training examples have been processed, one epoch is complete. The model's parameters (like weights and biases) continue to be refined over multiple epochs.

Process:

1. The model's parameters are usually initialized randomly
2. The model makes predictions on the training data
3. The loss or error is calculated based on the difference between predictions and true values
4. The model's parameters are updated using an optimization algorithm to minimize the loss
5. After processing the entire dataset once, one epoch is completed
6. The model's performance is typically evaluated on a separate validation dataset to monitor overfitting
7. The process is repeated for several epochs until a stopping condition is met (e.g., set number of epochs or no further improvement)

11.10 Steps for Neural Network Training and Optimization

Step 1: Preprocess Data
Prepare the data for optimal interaction between features and layers:

- Normalize or standardize inputs to ensure different features are on comparable scales
- Consider batch normalization to stabilize intermediate activations

Step 2: Initial Configuration
Choose an initial model configuration:

- Decide on architecture: number and size of layers
- Select loss function, optimizer, learning rate, and number of epochs
- Start with a simple model and gradually increase complexity as needed

Step 3: Initialize Weights and Biases
Properly initialize model parameters (weights and biases) to enable stable and efficient training from the start.

Step 4: Forward Pass and Backpropagation
For each training batch:

- Perform a forward pass to compute predictions and evaluate the loss
- Use backpropagation to compute gradients
- Update weights in a direction that reduces the loss

Step 5: Train the Network and Analyze Learning Curves
Monitor training and validation loss over time:

- Assess whether the model exhibits high bias (underfitting) or high variance (overfitting)
- Use these insights to guide subsequent model adjustments

Step 6: Hyperparameter Optimization and Model Adjustment
Based on the learning curves:

- If the model underfits: increase model capacity (more layers or larger layers), train longer
- If the model overfits: apply regularization techniques (dropout, weight decay, data augmentation)
- Use methods like k-fold cross-validation to systematically tune hyperparameters

Step 7: Retrain and Finalize the Model
Once optimal hyperparameters are selected:

- Retrain the model on the full training dataset
- Evaluate performance on a held-out test set before deployment

> **Note.** The training process is iterative. Observing the learning curves during training helps determine whether the model is in a high variance or bias regime, which guides subsequent hyperparameter optimization steps.

12 Lecture 11: Physics-Informed machine learning

12.1 Types of physics-informed machine learning

Three are 3 ways to incorporate physics into machine learning models:

- Physics-informed loss functions: Incorporate physical laws as constraints in the loss function to guide the learning process. (Soft constraints)
- Physics-informed architectures: Design neural network architectures that inherently respect physical principles (e.g., conservation laws, symmetries). (Hard constraints)
- Physics in features or data representation: Use physics-based features or representations of the data to improve model performance and interpretability.

Let's discuss the first one for a neural network

12.1.1 Physics-informed loss functions

In physics-informed machine learning, we can incorporate physical laws into the loss function to guide the training of the model. Some examples of physics-informed loss functions include:

- **Mechanistic modeling**: Incorporate known physical laws or equations directly into the loss function. For example, if a system is governed by a differential equation, the loss function can include a term that penalizes deviations from the equation.

  $$N(u(x,t), x, t, \lambda) = 0$$

  Here, N represents the physical law (e.g., a PDE), $u(x,t)$ is the unknown physical function, and λ are the parameters of the model. For example, the heat/diffusion equation can be represented as:

  $$\frac{\partial u}{\partial t} - D\frac{\partial^2 u}{\partial x^2} = 0$$

  Advantages: Now this type of loss function has the advantage of being interpretable and grounded in physical principles, it respects the underlying physics of the problem, and it can improve generalization

to unseen data.
Disadvantage: It may require a good understanding of the physical system and may not be applicable to all types of problems. Additionally, it can be computationally expensive to evaluate the physics-based loss term, especially for complex systems. It has unknown parameters (e.g., diffusion coefficient D) that need to be estimated from data, which can add complexity to the training process.

- Data-driven modeling: Use data to learn the underlying physical relationships without explicitly incorporating known equations. The loss function can be designed to encourage the model to learn patterns that are consistent with physical principles, such as smoothness or conservation laws.

  $$u(x,t) \simeq u_\theta(x,t)$$

  Here, $u_\theta(x,t)$ is the output of the neural network parameterized by θ. The loss function can include terms that encourage the model to learn physically consistent patterns, such as:

  $$L(\theta) = \frac{1}{N}\sum_{i=1}^{N}(u(x_i,t_i) - u_\theta(x_i,t_i))^2 \implies \min_\theta L(\theta)$$

  Advantages: This approach can be more flexible and can capture complex relationships in the data without requiring explicit knowledge of the underlying physics. It can also be more computationally efficient than mechanistic modeling, especially for high-dimensional problems. You also don't need to know the underlying physics of the problem, which can be beneficial for complex systems where the physics is not well understood.
  Disadvantages: It may require a large amount of data to learn the underlying physical relationships accurately, and it may not generalize well to unseen data if the training data is not representative of the true physical system. Additionally, it may be less interpretable than mechanistic modeling, as the learned relationships may not have a clear physical meaning.

12.2 Physics-informed neural network (PINNs)

One of the most common ways to incorporate physics into machine learning models is through Physics-Informed Neural Networks (PINNs). PINNs are a type of neural network that incorporates physical laws as constraints in the loss function. For the data $\{(t_i, x_i, u_i^{\text{obs}})\}_{i=1}^N$, we can define the loss function for a PINN as follows:

$$L_{(\text{data})}(\theta) = \sum_{i=1}^{N} (u_\theta(t_i, x_i) - u_i^{\text{obs}})^2$$

Now we know that the output of the neural network should not have a large deviation from the underlying physics,

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}$$

Now we can compute $\frac{\partial u_\theta}{\partial t}$ and $\frac{\partial^2 u_\theta}{\partial x^2}$ using automatic differentiation and back-propagation. The physics-informed loss term can be defined as:

$$L_{(\text{physics})}(\theta) = (|r_\theta(x_i, t_i)|)^2$$

Where,

$$r_\theta(x_i, t_i) = \frac{\partial u_\theta}{\partial t} - D \frac{\partial^2 u_\theta}{\partial x^2}$$

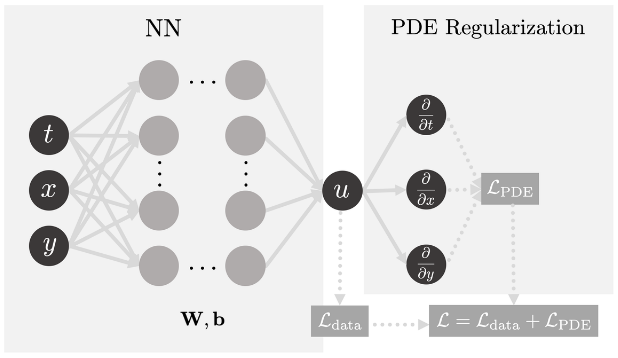

NN
PDE Regularization
t
x
y
u
$\frac{\partial}{\partial t}$
$\frac{\partial}{\partial x}$
$\frac{\partial}{\partial y}$
$\mathcal{L}_{\text{PDE}}$
$\mathbf{W}, \mathbf{b}$
$\mathcal{L}_{\text{data}}$
$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{PDE}}$

Figure 21: Physics-Informed Neural Networks (PINNs): Incorporating physical laws as constraints in the loss function

Warning

$r_\theta(x_i, t_i)$ will be zero if the neural network perfectly satisfies the underlying physics, and it will be non-zero if there are deviations from the physics. However, in practice, we can never satisfy the physics perfectly, so we want to minimize the physics-informed loss term to encourage the model to learn solutions that are consistent with the underlying physical principles.

In general, our DE might have intial and boundary conditions, so we can also include additional loss terms to enforce these conditions.

$$\text{DE}: \quad N(u(x,t), x, t, \lambda) = 0 \quad (x,t) \in \Omega \times [0,T]$$

$$\text{Initial condition}: \quad u(x,0) = u_0(x)$$

$$\text{Boundary condition}: \quad u(x,t) = g(x,t)$$

So the loss functions for the initial and boundary conditions can be defined as:

$$L_{(\text{initial})}(\theta) = \sum_{i=1}^{N_{\text{initial}}} (u_\theta(x_i, 0) - u_0(x_i))^2$$

$$L_{(\text{boundary})}(\theta) = \sum_{i=1}^{N_{\text{boundary}}} (u_\theta(x_i, t_i) - g(x_i, t_i))^2$$

The total loss function for training the PINN can be defined as a weighted sum of the data loss, physics-informed loss, initial condition loss, and boundary condition loss:

$$L(\theta) = \lambda_D L_{(\text{data})}(\theta) + \lambda_P L_{(\text{physics})}(\theta) + \lambda_I L_{(\text{initial})}(\theta) + \lambda_B L_{(\text{boundary})}(\theta)$$

Where λ_D, λ_P, λ_I, and λ_B are hyperparameters that control the relative importance of each loss term. By minimizing this total loss function, we can train the PINN to learn a solution that is consistent with both the observed data and the underlying physical principles.

Note. The terms λ_D, λ_P, λ_I, and λ_B can differ by order of magnitude, casuing an ill-conditioned optimization problem. To address this issue, we use

- Learing rate scheduling: Adjust the learning rates for each loss term during training to ensure that they are optimized effectively.

- Adaptive weighting: Dynamically adjust the weights of the loss terms based on their relative magnitudes during training to ensure that they contribute appropriately to the overall loss.
- Curriculum learning: Start with a simpler version of the problem (e.g., only data loss) and gradually introduce the physics-informed loss and other terms as training progresses.

12.2.1 Inverse problems with PINNs

Consider the following PDE with an unknown parameter λ:

$$N(u(x,t),x,t,\lambda) = 0$$

Now suppose we want to approximate the solution $u(x,t)$ using a neural network $u_\theta(x,t)$, and we also want to estimate the unknown parameter λ from data. For example,

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \implies r_{\text{pde}} = \frac{\partial u_\theta}{\partial t} - D\frac{\partial^2 u_\theta}{\partial x^2} \implies L_{(\text{physics})} = (|r_{\text{pde}}(x_i,t_i)|)^2$$

In this case, we can treat λ (e.g., D) as an additional parameter to be optimized during training. So we have 2 trainable parameters: θ (neural network parameters) and λ (unknown physical parameter). And our objective is to minimize the total loss function:

$$\min_{\theta,\lambda} L(\theta,\lambda) = \lambda_D L_{(\text{data})}(\theta) + \lambda_P L_{(\text{physics})}(\theta,\lambda) + \lambda_I L_{(\text{initial})}(\theta) + \lambda_B L_{(\text{boundary})}(\theta)$$

Nowe can calculate, $\frac{\partial L}{\partial \theta}$ and $\frac{\partial L}{\partial \lambda}$ using automatic differentiation and back-propagation, and update both θ and λ simultaneously during training,

$$\theta^{(k)} \leftarrow \theta^{(k-1)} - \eta_\theta \frac{\partial L}{\partial \theta}$$
$$\lambda^{(k)} \leftarrow \lambda^{(k-1)} - \eta_\lambda \frac{\partial L}{\partial \lambda}$$

12.2.2 Learning spatially varying parameters with PINNs

In some cases, the unknown parameter λ may not be a constant but rather a spatially varying function $\lambda(x)$. For example, consider the following PDE:

$$\frac{\partial u}{\partial t} = nabla\left(D(x)\frac{\partial u}{\partial x}\right) = D(x)\nabla^2 u + \nabla D(x)\cdot\nabla u$$

In this case, $D(x)$ is unknown. We can approximate $D(x)$ using another neural network $D_\phi(x)$, where ϕ are the parameters of this second neural network. So now we have two neural networks: $u_\theta(x,t)$ for approximating the solution and $D_\phi(x)$ for approximating the spatially varying parameter. The residual for the PDE can be defined as:

$$r_{\theta,\phi} = \frac{\partial u_\theta}{\partial t} - \frac{\partial}{\partial x}\left(D_\phi(x)\frac{\partial u_\theta}{\partial x}\right)$$

Now the optimization learns both the solution $u_\theta(x,t)$ and the spatially varying parameter $D_\phi(x)$.

Note. When learning $D(x)$ as a neural network, this increases the problem's complexity and may require more data and careful regularization to ensure that the learned function is physically meaningful and does not overfit the training data. For example, $\|\nabla D_\phi(x)\|^2$ to avoid overfitting and ensure smoothness of the learned spatially varying parameter.

Example. Suppose we want to solve the advection equation:

$$\frac{\partial u}{\partial t} + \beta\frac{\partial u}{\partial x} = 0$$

Now this works well for small β, but for large β, the solution becomes more complex. We can start with a small beta and train the PINN to learn the solution. Once we have a good solution for small beta, we can gradually increase beta and use the previously learned solution as a starting point for training the PINN with the larger beta. This is an example of curriculum learning, where we start with a simpler version of the problem and gradually increase its complexity to help the model learn effectively.

13 Lecture 12: Traning PINNs

13.1 Best practices for training PINNs

Consider the 1D diffusion equation:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial u}{\partial x}\right), \quad x \in \Omega \times [0,T]$$

$D(x)$ and $u(x,t)$ are unknown. Now we have 2 options to learn $D(x)$ and $u(x,t)$:

- **Option 1:** Use asingle network with 2 outputs. We can use a single neural network to predict both $u(x,t)$ and $D(x)$. The input to the network would be (x,t), and the output would be a vector containing both u and D.

  $$(u(x,t), D(x)) \simeq (u_\theta(x,t), D_\theta(x))$$

  The PDE residual can be defined as:

  $$r_\theta = \frac{\partial u_\theta}{\partial t} - \frac{\partial}{\partial x}\left(D_\theta(x)\frac{\partial u_\theta}{\partial x}\right)$$

  This approach is simple to implement, but it may be more difficult for the network to learn both the solution and the spatially varying parameter simultaneously, especially if they have different scales or complexities.

- **Option 2:** Use two separate networks. We can use one neural network to predict $u(x,t)$ and another neural network to predict $D(x)$. The first network takes (x,t) as input and outputs u, while the second network takes x as input and outputs D.

  $$u(x,t) \simeq u_\theta(x,t), \quad D(x) \simeq D_\phi(x)$$

  The PDE residual can be defined as:

  $$r_{\theta,\phi} = \frac{\partial u_\theta}{\partial t} - \frac{\partial}{\partial x}\left(D_\phi(x)\frac{\partial u_\theta}{\partial x}\right)$$

  This approach allows each network to specialize in learning its respective function, which can lead to better performance and easier training. However, it also increases the number of parameters and the complexity of the optimization problem.

However, notice that if $D(x)$ is constant, i.e, $D(x) = D$, then the PDE simplifies to:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}$$

In this case, the best practice would be treat D as a trainable scalar parameter. So a single network can be used to predict $u(x,t)$, and D can be optimized as a separate parameter during training. This approach is more efficient and easier to train than using two separate networks for u and D when D is constant.

$$\theta^{(new)} \leftarrow \theta^{(old)} - \eta_\theta \frac{\partial L}{\partial \theta}, \quad D^{(new)} \leftarrow D^{(old)} - \eta_D \frac{\partial L}{\partial D}$$

13.2 Gray-box modeling with PINNs

Definition 46. **Gray-box modeling** refers to a modeling approach that combines both mechanistic (physics-based) and data-driven components. In the context of PINNs, gray-box modeling involves using a neural network to learn the unknown or complex parts of a physical system while incorporating known physical laws or equations as constraints in the loss function. Basically we are learning the missing component/physics:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f_{\text{known}}(u, x, t) + g_{\text{unknown}}(u, x, t)$$

Example. FOr pendulum dynamics, we have the equation of motion:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{L}\sin(\theta)$$

Now suppose with friction, the equation becomes:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{L}\sin(\theta) - F_{friction}$$

Most common friction model is the linear damping model, where $F_{friction} = -c\frac{\mathrm{d}\theta}{\mathrm{d}t}$, but this may not be accurate for all types of friction. So we can use a neural network to learn the unknown friction term:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{L}\sin(\theta) + F_\phi(\theta, \frac{\mathrm{d}\theta}{\mathrm{d}t})$$

We approximate $\theta(t) = \theta_\psi(t)$ and $F_{friction} = F_\phi(\theta, \frac{\mathrm{d}\theta}{\mathrm{d}t})$ using two separate neural networks. The residual for the PDE can be defined as:

$$r_{\psi,\phi} = \frac{\mathrm{d}^2\theta_\psi}{\mathrm{d}t^2} + \frac{g}{L}\sin(\theta_\psi) - F_\phi(\theta_\psi, \frac{\mathrm{d}\theta_\psi}{\mathrm{d}t})$$

So the loss function for training the PINN can be defined as:

$$L(\psi, \phi) = \lambda_D L_{(\text{data})}(\psi) + \lambda_P L_{(\text{PDE})}(\psi, \phi)$$

13.3 Pre-training and transfer learning with PINNs

We pretrain a NN on syntheic data, and then we can fine-tune the pretrained model on the actual problem with real data. This approach can help the model learn useful features and representations from the synthetic data, which can improve its performance when fine-tuned on the real data.

$$\text{opertator}: \quad N_{\text{approx}}(u(x,t), x, t, \lambda) = 0$$

- **Pre-training:** We can generate synthetic/simulated data $D_{\text{sim}} = \{(x_i, t_i, u_i^{\text{sim}})\}_{i=1}^{N_s}$ using the the loss,

$$L_{\text{pretrain}}(\theta) = \sum_{i=1}^{N_s} (u_\theta(x_i, t_i) - u_i^{\text{sim}})^2$$

- **Fine-tuning:** After pre-training, we can fine-tune the model on real (measured) data $D_{\text{real}} = \{(x_i, t_i, u_i^{\text{exp}})\}_{i=1}^{N_r}$ using the loss,

$$L_{\text{real}}(\theta) = \sum_{i=1}^{N_r} (u_\theta(x_i, t_i) - u_i^{\text{exp}})^2$$

  We can adjust some of the weights, or for example, we can ass a layer and only train this new layer.

For a physics based NN,
Generated data $\rightarrow$ Pre-trained NN $\rightarrow$ Fine-tuned NN on real data.

> **Note.** Suppose the original physics model is exact but complicated. Then,
> Generate Data $\rightarrow$ train a NN $\rightarrow$ dimension reduction.

13.4 Uncertainty in PINNs

13.4.1 Bayesian Neural Networks

First we discuss "Bayesian Neural Networks" (BNN). In standard NN, we have one specific set of parameters θ^*. This gives us the same prediction for a given input (after training):

$$u(x_i, t_i) \simeq u_{\theta^*}(x_i, t_i)$$

The key idea in BNN is to treat weights, as random variables (not fixed). We learn a distribution for θ.

$$\theta \sim P(\theta | \mathcal{D})$$

Where $\mathcal{D}$ is the observed data and P is the posterior.

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})}$$

$P(D|\theta)$ is the likelihood, $P(\theta)$ is the prior, and $P(D)$ is the evidence. On a high level, we are doing MCMC sampling to get samples from the posterior distribution of the parameters. Then we can use variational inference to approximate the posterior distribution with a simpler distribution (e.g., Gaussian). This allows us to capture uncertainty in the model parameters and make probabilistic predictions.

13.4.2 Bayesian PINNs

Now for Deterministic PINN, one best fit solution is learned u_{θ^*}. For Bayesian PINN, we learn a distribution over the solution space $P(u|\mathcal{D})$. This allows us to quantify uncertainty in the predictions and capture the variability in the data and the underlying physics.

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})}$$

The noise is given by,

$$\epsilon_i = \text{noise} \sim N(0, \sigma_d^2)$$

The data is,

$$\mathcal{D} = \{(x_i, t_i, y_i)\}_{i=1}^N$$

Where,

$$y_i = u_\theta(x_i, t_i) + \epsilon_i$$

So the likelihood can be defined as,

$$P(\mathcal{D}|\theta) \sim e^{-\frac{1}{2\sigma_d^2}\sum_{i=1}^N (u_\theta(x_i,t_i)-y_i)^2}$$

The prior can be defined as,

$$P(\theta) = e^{-\lambda L_{(\text{physics})}(\theta)} \sim e^{-\frac{1}{2\sigma_r^2}\|r_\theta\|^2}$$

Where σ_r^2 is the physics residual variance. We can also add a regularization term to the prior to prevent overfitting,

$$P(\theta) = e^{-\lambda L_{(\text{physics})}(\theta)} e^{-\frac{1}{2\sigma_\theta^2}\|\theta\|^2}$$

So the loss function for training the Bayesian PINN can be defined as,

$$L(\theta) = -\log P(\theta|\mathcal{D}) = -\frac{1}{2\sigma_d^2}\sum_{i=1}^{N}(u_\theta(x_i,t_i) - y_i)^2 + \frac{1}{2\sigma_r^2}\|r_\theta\|^2 + \frac{1}{2\sigma_\theta^2}\|\theta\|^2$$

Where σ_d^2 is the data noise variance, σ_r^2 is the physics residual variance, and σ_θ^2 is the regularization variance.
Next, we can use MCMC sampling to sample:

$$\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(S)} \sim P(\theta|\mathcal{D})$$

Then for each sample,

$$u^{(s)}(x,t) = u_{\theta^{(s)}}(x,t)$$

Finally, we can compute the mean and variance of the predictions to quantify uncertainty:

$$\bar{u}(x,t) = \frac{1}{S}\sum_{s=1}^{S} u^{(s)}(x,t)$$

$$\mathrm{Var}(u(x,t)) = \frac{1}{S}\sum_{s=1}^{S}(u^{(s)}(x,t) - \bar{u}(x,t))^2$$