МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

ЛЕКЦІЯ 2. НАВЧАННЯ З УЧИТЕЛЕМ: РЕГРЕСІЯ, КЛАСИФІКАЦІЯ ТА ОПТИМІЗАЦІЯ

Львів -- 2026

Лекція зі штучного інтелекту 2026-02

Вступ

На попередній лекції ми оглянули парадигми штучного інтелекту та познайомилися з таксономією машинного навчання. Тепер настав час заглибитися у найпоширенішу парадигму — **навчання з учителем** (*supervised learning*) — і зрозуміти, як воно працює «під капотом».

Навчання з учителем — це задача побудови моделі, яка вчиться передбачувати відповідь на основі прикладів. Уявіть, що ви — геймдизайнер, і хочете автоматично оцінювати складність рівня за його параметрами: кількість ворогів, розмір арени, час на проходження. Ви збираєте дані від тестерів — для кожного рівня знаєте параметри і реальну оцінку складності. Навчання з учителем дозволяє побудувати модель, яка навчиться цей зв'язок відтворювати.

У цій лекції ми розглянемо два основні типи задач — регресію та класифікацію, — вивчимо, як модель оцінює свої помилки (функції втрат), як вона покращує себе (градієнтний спуск) та як правильно налаштовувати параметри навчання.

Теми, що розглядаються

1. Математична постановка задачі навчання з учителем
2. Лінійна регресія та функція втрат MSE
3. Градієнтний спуск: batch, stochastic, mini-batch
4. Логістична регресія, межі прийняття рішень та порівняння з SVM
5. Функція втрат крос-ентропія та метрики класифікації
6. Стратегії підбору гіперпараметрів
7. Ігрові застосування

Математична постановка задачі

Задача навчання з учителем формулюється так: у нас є набір даних (*dataset*) із N прикладів:

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$$

де x_i — вектор ознак (*features*), а y_i — цільова змінна (*target*), тобто «правильна відповідь».

Кожен приклад x_i — це точка у багатовимірному **просторі ознак** (*feature space*). Якщо ми описуємо гравця двома ознаками (час гри, кількість покупок), то кожен гравець — це точка на площині. Якщо ознак три — точка у тривимірному просторі. Для n ознак — точка у n-вимірному просторі, який ми вже не можемо візуалізувати, але математика працює однаково. Простір ознак — це фундаментальна абстракція ML: усі алгоритми, від лінійної регресії до нейронних мереж, працюють саме з точками у цьому просторі.

Наша мета — знайти функцію $f_\theta(x)$ з параметрами θ, яка для нового, раніше не баченого входу x дасть якомога точніше передбачення $\hat{y} = f_\theta(x)$.

Для програміста це можна уявити як пошук правильних коефіцієнтів у формулі. Наприклад, якщо ви хочете передбачити час проходження рівня (числова величина) — це **регресія**. Якщо потрібно визначити, чи залишиться гравець у грі через місяць (так/ні) — це **класифікація**.

Як зрозуміти, наскільки добре працює модель? Для цього використовують **функцію втрат** (*loss function*) $\mathcal{L}(\theta)$, яка вимірює різницю між передбаченнями моделі та реальними відповідями. Назва «втрати» (*loss*) не випадкова: вона відображає ідею ціни помилки. Кожна неточність моделі — це «втрата» якості, і чим більша різниця між передбаченням і реальністю, тим більша ця втрата. У деяких контекстах використовують також терміни **функція вартості** (*cost function*) — коли акцент на сумарній вартості помилок по всьому набору даних, або **цільова функція** (*objective function*) — коли говорять про оптимізацію загалом. На практиці ці терміни часто використовують як синоніми. Процес навчання — це мінімізація функції втрат:

$$\theta^* = \arg\min_\theta \mathcal{L}(\theta)$$

Тобто ми шукаємо такі параметри θ^*, при яких модель помиляється найменше.

Лінійна регресія

Ідея

Лінійна регресія — це найпростіша модель, яка шукає лінійну залежність між ознаками та цільовою змінною. Якщо ви коли-небудь будували лінію тренду в Excel-таблиці — ви вже використовували лінійну регресію.

Для одної ознаки формула виглядає як рівняння прямої:

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

де w (*weight*) — нахил прямої (вага), b (*bias*) — зсув. Термін **bias** у ML має специфічне значення, відмінне від побутового «упередження». У контексті моделі bias — це параметр зсуву, який дозволяє прямій (або гіперплощині) не проходити через початок координат. Без нього модель була б обмежена: пряма завжди проходила б через точку (0, 0), що рідко відповідає реальним даним. Зверніть увагу: в ML є й інше значення цього слова — **bias моделі** як систематична помилка через надто прості припущення (наприклад, спроба описати параболу прямою лінією). Ці два значення не слід плутати. Для кількох ознак формула узагальнюється до:

$$\hat{y} = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b = \mathbf{w}^T \mathbf{x} + b$$

Ігровий приклад. Ви хочете передбачити шкоду (*damage*) від удару у RPG-грі на основі рівня гравця x_1, сили зброї x_2 та модифікатора навички x_3:

$$\hat{y}_{damage} = w_1 \cdot level + w_2 \cdot weapon\_power + w_3 \cdot skill\_mod + b$$

Модель сама підбере ваги w_1, w_2, w_3 та зсув b, щоб найкраще відповідати реальним даним бою.

Функція втрат: середньоквадратична помилка (MSE)

Як оцінити якість лінійної регресії? Інтуїтивно — порівняти передбачення з реальністю і покарати модель за різницю. **MSE** (*Mean Squared Error*) — це середнє значення квадратів помилок:

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

Чому квадрат? По-перше, він робить усі помилки додатними. По-друге, великі помилки штрафуються непропорційно сильніше — це бажана властивість, бо великий промах зазвичай гірший за кілька маленьких.

Для розробника ігор MSE — як метрика якості балансу: якщо ваша формула шкоди регулярно відхиляється від очікуваної на 50 одиниць — це проблема, і квадрат це підкреслить.

Поліноміальні ознаки та нелінійна регресія

Лінійна регресія обмежена прямими лініями (або площинами у багатовимірному випадку). Але реальні залежності часто нелінійні. Наприклад, шкода від падіння в грі зростає не лінійно з висотою — повільно для малих висот і різко для великих.

Елегантний прийом: замість зміни самого алгоритму можна **розширити простір ознак**, створивши нові ознаки з існуючих. Якщо у нас є ознака x, ми можемо додати x^2, x^3 і навіть комбінації на кшталт $x_1 \cdot x_2$. Такі штучно створені ознаки називаються **поліноміальними ознаками** (*polynomial features*).

Приклад: маючи одну ознаку — висоту падіння h, — ми створюємо розширений вектор ознак:

$$x \to [h,\ h^2,\ h^3]$$

Тепер лінійна регресія на цих нових ознаках стає **поліноміальною регресією**:

$$\hat{y} = w_1 h + w_2 h^2 + w_3 h^3 + b$$

Зверніть увагу: модель залишається лінійною відносно *параметрів* w_1, w_2, w_3 — ми все ще використовуємо зважену суму. Але відносно *вхідних даних* h вона тепер нелінійна — це крива, а не пряма. Ми не змінили алгоритм навчання — лише трансформували дані, перемістивши їх у новий, більш виразний простір ознак.

Цей прийом — **розширення простору ознак** (*feature expansion*) — є надзвичайно потужною ідеєю, яка з'явиться знову, коли ми будемо говорити про ядрові функції у методі опорних векторів (SVM) далі в цій лекції.

Градієнтний спуск

Інтуїція

Маючи функцію втрат, нам потрібен метод її мінімізації. **Градієнтний спуск** (*gradient descent*) — це ітеративний алгоритм, який крок за кроком зменшує значення функції втрат.

Уявіть, що ви стоїте на горі в густому тумані і хочете спуститися в долину. Ви не бачите всієї гори, але можете відчути нахил під ногами. Стратегія проста: на кожному кроці йдіть в напрямку найбільшого спуску. Саме це і робить градієнтний спуск.

Математично: **градієнт** $\nabla_\theta \mathcal{L}$ — це вектор часткових похідних функції втрат за кожним параметром. Він вказує напрямок найшвидшого зростання функції. Щоб мінімізувати, ми рухаємось у протилежному напрямку:

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta \mathcal{L}$$

де α — **швидкість навчання** (*learning rate*) — розмір кроку.

Швидкість навчання

Параметр α критично важливий:

- **Занадто великий** α — модель «перестрибує» мінімум, значення функції втрат починає коливатися або навіть зростати. Це як бігти з гори занадто швидко — промахнетеся повз долину.
- **Занадто малий** α — модель сходиться нестерпно повільно. Тисячі ітерацій можуть не принести суттєвого прогресу.
- **Правильний** α — зазвичай підбирається експериментально. Типові початкові значення: 0.01, 0.001, 0.0001.

Три варіанти градієнтного спуску

Різниця між варіантами — у тому, скільки прикладів використовується для обчислення градієнта на кожному кроці.

Batch Gradient Descent обчислює градієнт на *всьому* наборі даних. Це дає точний напрямок руху, але на великих даних один крок може тривати дуже довго. Якщо у вас мільйон записів ігрової телеметрії — чекати обчислення градієнта по всьому мільйону для кожного кроку непрактично.

Stochastic Gradient Descent (SGD) обчислює градієнт на *одному* випадковому прикладі. Кожен крок дуже швидкий, але напрямок «шумний» — градієнт, обчислений по одному прикладу, може сильно відрізнятися від справжнього. Модель рухається зигзагами, але в середньому — в правильному напрямку. Цей шум іноді навіть корисний: він допомагає вибратися з поганих локальних мінімумів.

Mini-batch Gradient Descent — золота середина. Градієнт обчислюється на невеликому пакеті (*batch*) із B прикладів (типово 32, 64, 128 або 256). Це дає достатньо точний напрямок і при цьому ефективно використовує паралелізм GPU. Саме цей варіант використовується на практиці у переважній більшості сучасних ML-систем.

Варіант	Приклади на крок	Швидкість кроку	Точність градієнта	Використання GPU
Batch	Всі N	Повільно	Висока	Не ефективно

Stochastic	1	Дуже швидко	Низька	Не ефективно
Mini-batch	B (32–256)	Швидко	Достатня	Ефективно

Логістична регресія та класифікація

Від регресії до класифікації

Лінійна регресія передбачає числове значення. Але що, якщо нам потрібно передбачити категорію? Наприклад, чи піде гравець із гри (*churn*: так/ні)? Тоді потрібна **класифікація**.

Ключова ідея **логістичної регресії** — взяти лінійну модель і пропустити її результат через функцію, що перетворює будь-яке число в ймовірність від 0 до 1. Ця функція називається **сигмоїда** (*sigmoid*):

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \text{де } z = \mathbf{w}^T\mathbf{x} + b$$

Сигмоїда має характерну S-подібну форму: для дуже від'ємних z вона дає значення близько 0, для дуже додатних — близько 1, а при $z = 0$ повертає рівно 0.5.

Результат $\sigma(z)$ інтерпретується як ймовірність належності до позитивного класу. Якщо $\sigma(z) \geq 0.5$ — передбачаємо клас 1, інакше — клас 0.

Межа прийняття рішень (Decision Boundary)

Логістична регресія фактично шукає лінійну межу (*decision boundary*), яка розділяє простір ознак на два класи. Точки по один бік від межі класифікуються як клас 0, по інший — як клас 1.

Ігровий приклад. Уявіть двовимірний простір: вісь X — середній час сесії гравця, вісь Y — кількість внутрішньоігрових покупок. Логістична регресія проведе пряму лінію, що розділить гравців на тих, хто залишиться (зелені точки), і тих, хто піде (червоні точки). Усі точки «вище» прямої — активні гравці, «нижче» — потенційний відтік.

Обмеження: якщо реальна межа між класами нелінійна (наприклад, дугоподібна), проста логістична регресія не зможе її точно змоделювати — для цього потрібні складніші моделі.

Порівняння з методом опорних векторів (SVM)

Логістична регресія — не єдиний метод лінійної класифікації. **Метод опорних векторів** (*Support Vector Machine*, SVM) розв'язує ту саму задачу — знаходить межу між класами — але з іншою філософією.

Логістична регресія шукає межу, яка максимізує ймовірність правильної класифікації всіх прикладів. SVM шукає межу з **максимальним відступом** (*margin*) — тобто таку, що знаходиться якомога далі від найближчих точок обох класів. Ці найближчі точки називаються **опорними векторами** (*support vectors*) — саме вони визначають положення межі.

Аналогія: уявіть, що ви прокладаєте дорогу між двома селами (класами) на карті. Логістична регресія проведе дорогу так, щоб загалом було зручно для всіх мешканців. SVM проведе дорогу якомога далі від крайніх будинків обох сіл — максимізуючи «буферну зону».

На практиці для лінійно розділюваних даних обидва методи дають схожі результати. Але SVM має важливу перевагу: **ядрові функції** (*kernel trick*). Пригадайте, як у розділі про регресію ми розширювали простір ознак поліноміальними ознаками — брали x і створювали $[x, x^2, x^3]$. Це дозволяло лінійній моделі описувати нелінійні залежності. Ядрова функція — це той самий принцип, доведений до елегантної крайності: SVM *неявно* працює у просторі значно вищої розмірності, де дані стають лінійно розділюваними, але при цьому не потребує явного обчислення всіх нових ознак. Ядро обчислює лише скалярний добуток між точками у цьому розширеному просторі, що робить метод обчислювально ефективним навіть для нескінченновимірних просторів ознак.

Основні типи ядер:

- **Поліноміальне ядро** (*polynomial kernel*) — відповідає явному створенню поліноміальних ознак до певного степеня. Наприклад, ядро степеня 2 для двох ознак (x_1, x_2) неявно працює з простором $[x_1, x_2, x_1^2, x_2^2, x_1 x_2]$
- **RBF-ядро** (*Radial Basis Function*) — відповідає нескінченновимірному простору ознак, дозволяючи будувати довільно складні криволінійні межі. Це найпопулярніше ядро на практиці
- **Лінійне ядро** — еквівалентне звичайній лінійній класифікації без розширення

Властивість	Логістична регресія	SVM
Результат	Ймовірність класу (0–1)	Відстань до межі (без ймовірності)
Принцип	Максимум ймовірності	Максимальний відступ
Нелінійність	Тільки лінійна межа	Нелінійна через ядра
Масштабованість	Добре масштабується	Повільніше на великих даних
Інтерпретація	Легко інтерпретувати	Складніше інтерпретувати

Для задач у іграх вибір між ними залежить від контексту: якщо потрібна ймовірність (наприклад, «гравець піде з ймовірністю 73%»), логістична регресія зручніша. Якщо потрібна максимальна точність класифікації на невеликому наборі даних із складною межею — SVM з ядром може бути кращим вибором. У сучасній практиці обидва методи часто поступаються нейронним мережам на великих даних, але залишаються відмінним базовим рішенням (*baseline*) і чудовим інструментом для швидкого прототипування.

Багатокласова класифікація: Softmax

Для задач з більш ніж двома класами (наприклад, передбачити тип гравця: «casual», «hardcore», «social») використовується узагальнення — функція **Softmax**:

$$P(y = k|\mathbf{x}) = \frac{e^{zk}}{\sum_{j=1}^{K} e^{zj}}$$

де z_k — оцінка (*score*) для класу k, а K — кількість класів. Softmax перетворює вектор довільних чисел у вектор ймовірностей, що сумуються до 1.

Функція втрат: крос-ентропія

Чому не MSE для класифікації?

Для задач класифікації MSE працює погано. Причина: якщо ми використовуємо сигмоїду, MSE створює функцію втрат з «плоскими» ділянками, де градієнт майже нульовий — модель перестає навчатися, хоча ще далека від правильної відповіді.

Натомість використовується **крос-ентропія** (*cross-entropy loss*), яка штрафує модель тим сильніше, чим впевненіше вона помиляється.

Бінарна крос-ентропія

Для двох класів (наприклад, churn/no churn):

$$\mathcal{L}_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\Big]$$

Тут $\hat{y}_i = \sigma(\mathbf{w}^T\mathbf{x}_i + b)$ — передбачена ймовірність, $y_i \in \{0, 1\}$ — справжня мітка.

Інтуїція: якщо правильна відповідь $y = 1$, а модель передбачила $\hat{y} = 0.99$ — штраф мінімальний ($-\log 0.99 \approx 0.01$). Але якщо модель передбачила $\hat{y} = 0.01$ — штраф величезний ($-\log 0.01 \approx 4.6$). Модель «боляче» карається за впевнені помилки.

Категоріальна крос-ентропія

Для багатьох класів:

$$\mathcal{L}_{CE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{i,k}\log(\hat{y}_{i,k})$$

де $y_{i,k}$ — one-hot вектор справжнього класу, $\hat{y}_{i,k}$ — softmax-ймовірність класу k. Саме ця функція втрат використовується для тренування класифікаторів зображень, включаючи ResNet з першої лабораторної роботи.

Підбір гіперпараметрів

Параметри vs гіперпараметри

У машинному навчанні розрізняють два типи параметрів:

- **Параметри моделі** (θ: ваги w, зсуви b) — навчаються автоматично під час тренування. Їх може бути тисячі або мільйони.
- **Гіперпараметри** — налаштовуються розробником *до* початку тренування: швидкість навчання, розмір батчу, кількість епох, архітектура моделі тощо.

Для розробника ПЗ різниця така: параметри — це те, що оптимізує алгоритм; гіперпараметри — це конфігурація самого алгоритму, яку ви задаєте в коді перед запуском.

Основні гіперпараметри

Гіперпараметр	Що контролює	Типові значення
Learning rate (α)	Розмір кроку оптимізації	10^{-4} — 10^{-1}
Batch size	Кількість прикладів у батчі	32, 64, 128, 256
Кількість епох	Скільки разів переглянути весь набір даних	10 — 100+
Регуляризація	Запобігання перенавчанню	L2: 10^{-4}, dropout: 0.1–0.5

Епоха (*epoch*) — один повний прохід усього навчального набору даних. Тобто якщо у вас 10,000 прикладів і batch size 100, то одна епоха = 100 кроків оптимізації.

Стратегії підбору

Grid Search — перебір усіх комбінацій значень на заздалегідь визначеній сітці. Наприклад, learning rate $\in$ {0.001, 0.01, 0.1} × batch size $\in$ {32, 64} дає 6 комбінацій. Кожна комбінація — повний цикл тренування і оцінки. Простий, але дорогий метод.

Random Search — випадковий вибір комбінацій гіперпараметрів. Дослідження (Bergstra & Bengio, 2012) показало, що Random Search часто ефективніший за Grid Search при тому самому бюджеті обчислень, оскільки різні гіперпараметри мають різну важливість, і випадковий пошук краще досліджує важливі виміри.

Learning Rate Schedule — зменшення швидкості навчання під час тренування. Поширені стратегії: Step Decay (зменшення вдвічі кожні k епох), Cosine Annealing (плавне зменшення за косинусоїдою). Інтуїція: спочатку робимо великі кроки, щоб швидко наблизитися до мінімуму, потім маленькі — для точного налаштування.

Валідація та перенавчання

Щоб оцінити якість моделі, дані розбивають на три частини:

- **Тренувальний набір** (training set, ~70–80%) — на ньому модель навчається
- **Валідаційний набір** (validation set, ~10–15%) — для підбору гіперпараметрів
- **Тестовий набір** (test set, ~10–15%) — фінальна оцінка, яку модель «бачить» лише один раз

Перенавчання (*overfitting*) — ситуація, коли модель ідеально вивчила тренувальні дані, але погано працює на нових. Аналогія: студент, що вивчив відповіді на конкретні задачі з підручника, але не розуміє принципів і не може розв'язати нову задачу.

Ознаки перенавчання: training loss падає, а validation loss починає зростати.

Ігрові застосування

Передбачення складності рівня

Задача: маючи параметри рівня (кількість ворогів, площа арени, кількість укриттів, час на проходження), передбачити суб'єктивну складність (1–10) для середнього гравця.

Підхід: лінійна регресія або поліноміальна регресія з MSE як функцією втрат.

Дані: записи плейтестерів, де кожен запис — вектор параметрів рівня + оцінка складності.

Користь для розробника: автоматичний скринінг рівнів при процедурній генерації. Замість того щоб тестувати кожен згенерований рівень вручну, модель може миттєво оцінити його складність і відфільтрувати занадто прості або складні варіанти.

Моделювання відтоку гравців (Player Churn)

Задача: передбачити ймовірність того, що гравець припинить грати протягом наступних 7 днів.

Підхід: логістична регресія (бінарна класифікація) з крос-ентропією.

Ознаки: частота сесій за останній тиждень, середня тривалість сесії, кількість покупок, прогрес у грі, соціальна активність (гільдія, друзі).

Користь: визначивши гравців з високим ризиком відтоку, можна вчасно запропонувати їм бонуси, нові квести або персоналізований контент.

Як оцінити якість класифікатора відтоку? Проста точність (*accuracy* — частка правильних передбачень) тут оманлива. Якщо лише 5% гравців йдуть, модель, що завжди передбачає «залишиться», матиме accuracy 95% — але буде абсолютно марною. Тому для задач класифікації з незбалансованими класами використовують додаткові метрики:

- **Precision** (точність) — серед усіх гравців, яких модель позначила як «піде», яка частка справді пішла? Висока precision означає, що ви не турбуєте лояльних гравців зайвими бонусами. Формально: $\text{Precision} = \frac{TP}{TP+FP}$
- **Recall** (повнота) — серед усіх гравців, які справді пішли, яку частку модель виявила? Високий recall означає, що ви не пропускаєте гравців, які збираються піти. Формально: $\text{Recall} = \frac{TP}{TP+FN}$
- **F1-score** — гармонійне середнє precision і recall: $F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$. Корисний, коли потрібен один показник, що враховує обидва аспекти.

Тут TP (*true positives*) — правильно виявлений відтік, FP (*false positives*) — хибна тривога (модель сказала «піде», але гравець залишився), FN (*false negatives*) — пропущений відтік (модель сказала «залишиться», але гравець пішов).

На практиці завжди є компроміс: збільшуючи recall (виявляємо більше реального відтоку), ми зазвичай знижуємо precision (більше хибних тривог). Вибір балансу залежить від бізнес-контексту: якщо вартість бонусу для гравця низька, краще мати високий recall; якщо бонуси дорогі — пріоритет precision.

Оцінка шкоди та фізичне моделювання

Задача: створити модель для передбачення результату фізичної взаємодії в грі: відстань польоту снаряда, шкода від зіткнення, траєкторія уламків.

Підхід: лінійна або поліноміальна регресія, тренована на симульованих або записаних даних.

Ознаки: маса об'єкта, швидкість, кут удару, тип матеріалу.

Користь: ML-модель може замінити складні фізичні обчислення в реальному часі. Якщо повна фізична симуляція є занадто дорогою для обчислення (наприклад, деформація автомобіля при зіткненні), натренована модель може давати візуально правдоподібні результати за мікросекунди замість мілісекунд.

Зв'язок з лабораторними роботами

У лабораторній роботі №1 ви вже використовуєте навчання з учителем, навіть якщо не тренуєте модель з нуля:

- **ResNet з передавальним навчанням** — класифікація зображень. Під час fine-tuning ви мінімізуєте крос-ентропію за допомогою оптимізатора Adam (вдосконалений варіант SGD)
- **DataLoader** — завантажує дані mini-batch'ами, тобто ви використовуєте Mini-batch Gradient Descent
- **Learning Rate Scheduler** — зменшення α під час тренування

У подальших лабораторних роботах ми будемо явно експериментувати з різними функціями втрат, стратегіями оптимізації та гіперпараметрами.

Висновок

У цій лекції ми розглянули фундаментальні компоненти навчання з учителем:

- **Лінійна регресія** з MSE — для передбачення числових величин
- **Логістична регресія** з крос-ентропією — для класифікації, та її порівняння з SVM
- **Градієнтний спуск** — механізм навчання, і чому mini-batch є стандартом індустрії
- **Метрики класифікації** — precision, recall, F1-score та їх значення для практичних задач
- **Гіперпараметри** — конфігурація навчання, яку підбирає розробник
- **Ігрові застосування** — від передбачення складності до моделювання відтоку гравців

Ці концепції лежать в основі всього сучасного машинного навчання. Навіть найскладніші нейронні мережі тренуються тим самим принципом: визначити функцію втрат, обчислити градієнт, зробити крок оптимізації.

На наступній лекції ми перейдемо до нейронних мереж: як з'єднання простих лінійних моделей у шари створює потужні нелінійні моделі, і як регуляризація запобігає перенавчанню.