מיסודות למימוש - מדריך מעשי
למהנדסים ומפתחים

יסודות הלמידה העמוקה המודרנית

מהדורה 1.1 | ספטמבר 2025

ד"ר ברק אור

metaor.ai

יסודות הלמידה העמוקה המודרנית

זה לא עוד ספר על למידה עמוקה.
יסודות הלמידה העמוקה המודרנית הוא מדריך יישומי לבנייה, אימון ומימוש של רשתות ניורונים עמוקות – נכתב במיוחד בעברית - עבור מהנדסים שמתעניינים במערכות בעולם האמיתי ולא רק בתוצאות תאורטיות.
הספר כולל CNNs, LSTMs, Transformers, Autoencoders ועוד. הוא מכסה אימון, אופטימיזציה (AdamW, Lion), Mixed Precision ו־Regularization. בנוסף מוסברים Explainability (SHAP, Grad-CAM), Time Series, Tabular Data ו־Transfer Learning.
התוכן תמציתי ובמיקוד הנדסי - מותאם לאנשי מקצוע. הספר משמש גם כרשימות קורס בהכשרה עצמית של ArtificialGate. כל החומרים פותחו כדי לתמוך בלומדים מרקעים טכנולוגיים שונים וזמינים במספר שפות.

ד"ר ברק אור הוא חוקר, מרצה ויזם בתחום הבינה המלאכותית. בעל דוקטורט בלמידת מכונה ומערכות ניווט, ושלושה תארים מהטכניון. מייסד ArtificialGate מבית metaor.ai. יועץ AI לתעשיות הביטחוניות ולחברות המובילות במשק הישראלי. החל מ-2024 משמש כמנהל האקדמי של המסלול לפיתוח בינה מלאכותית ולמידה עמוקה בבית הספר להייטק של Google ואוניברסיטת רייכמן.

יסודות הלמידה העמוקה המודרנית

ד"ר ברק אור

ספטמבר 2025

ArtificialGate

תוכן העניינים

מבוא

ברוכים הבאים.

ספר זה הוא תוצאה של שנים של עבודה מעשית בלמידה עמוקה - הדרכת מהנדסים, ליווי חברות והטמעת מודלים הפועלים בהיקף גדול. הוא מבוסס על הקורס היסודי שאני מלמד במסגרת ArtificialGate ונבנה בקפידה כך שייקח אתכם מאפס ועד ליישום מלא.

שמי ד"ר ברק אור. אני בעל דוקטורט בלמידת מכונה למערכות ניווט. בעשור האחרון ייסדתי סטארטאפים ושילבתי למידה עמוקה במערכות אמיתיות.

מידע נוסף: www.barakor.com.

ספר זה אינו סקירה מחקרית ואינו דיון תאורטי. הוא בסיס למהנדסים ונועד לתת לכם:

- להבין איך למידה עמוקה עובדת מהבסיס
- לכתוב, לאמן ולהעריך מודלים ב־PyTorch, Tensorflow
- לשמור ולממש מודלים בסביבות דמויות־פרודקשן
- לבחור כלים, פורמטים ופרקטיקות עדכניות לעבודה ב־Deep Learning

אודות המחבר

ד"ר ברק אור הוא יזם, חוקר ומרצה המתמחה בתחום Artificial Intelligence, עם התמחות ב־Deep Learning וב־Machine Learning-based Inertial Sensing.

עבודתו הובילה לרישום מספר פטנטים, לפרסומים בכתבי עת ולמחקר יישומי באקדמיה ובתעשייה. מאז 2024, ד"ר אור משמש כמנהל האקדמי של מסלול פיתוח מודלי למידה עמוקה ו-AI בבית הספר להייטק של Google ואוניברסיטת רייכמן. דרכו בהוראה החלה כמתרגל בטכניון וכמורה לפיזיקה בבית הספר הריאלי העברי בחיפה ובהמשך התרחבה להכשרות מתקדמות בתחום ה־DL למהנדסים ולחוקרים באקדמיה ובתעשיה.

ד"ר אור הוא מייסד ArtificialGate, פלטפורמה אינטראקטיבית מקוונת ללימוד AI, המיועדת לקהלים טכנולוגיים וארגוניים וכן ייסד מספר מיזמים נוספים בתחום Applied AI. בין אלה ניתן למנות את metaor.ai, מעבדת מחקר והכשרה; Alma Technologies, סטארט־אפ Deep-Tech עבור פתרונות מיקום לרכב מבוססי חיישנים אינרציאליים לסביבות נטולות GPS; ואת AIME Systems, העוסקת ביישום AIבתחום בריאות האישה. עבודתו בתחומים אלה זכתה למענקים מרשות החדשנות, ממפא"ת וממשרד הכלכלה והתעשייה והניבה גם מספר פטנטים.

בשלבים מוקדמים יותר בקריירה שלו, זכה ד"ר אור בפרס גמונדר והגיע למקום השני בתחרות החדשנות הביטחונית של הטכניון. הרקע האקדמי שלו כולל דוקטורט ב־Machine Learning-based Inertial Navigation מאוניברסיטת חיפה (2022), תואר M.Sc. בהנדסת אווירונאוטיקה וחלל מהטכניון (2018), תואר B.Sc. בהנדסת אווירונאוטיקה וחלל מהטכניון (2016) ותואר B.A. בכלכלה וניהול (בהצטיינות, 2016) מהטכניון.

שיעור 1

למידת מכונה לעומת למידה עמוקה

מבוא

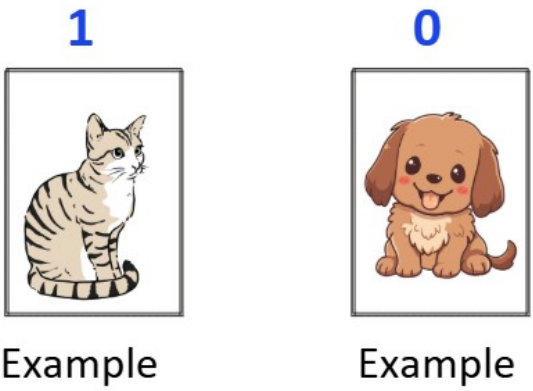
לפני שנבין איך נראית רשת נוירונים, או מה פירוש Gradient Descent, חשוב לבנות בסיס: מהי בעצם למידת מכונה? מה מבדיל אותה מלמידה עמוקה? ולמה ההבחנה הזו קריטית למהנדסים, לחוקרים ולמפתחים?

למידת מכונה: עקרונות בסיסיים

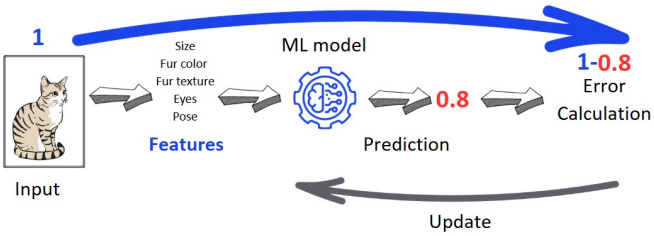
במקום לכתוב חוקים מפורשים, כמו: "אם במייל כתוב שזכית במיליון דולר - סמן אותו כ־Spam", אנו מספקים למודל אלפי מיילים מתויגים כתקינים או כ־Spam והמודל לומד בעצמו מה מבדיל בין שתי הקטגוריות.

כל מודל כזה עובר תהליך המכונה "אימון" (Training). ב־Supervised Learning, אנו מספקים זוגות של קלטים ותוויות. למשל: תמונה של חתול מתויגת כ־"1" ותמונה של כלב מתויגת כ־"0". המודל מייצר חיזוי על סמך הקלט בלבד, משווה את החיזוי לתווית האמיתית, מחשב את השגיאה ומעדכן את עצמו כדי למזער את השגיאה. תהליך זה חוזר על עצמו אלפי פעמים - עד שהמודל לומד את הקשר בין הקלט (התמונה) לבין הפלט הרצוי.

במהלך האימון, המודל לומד לשייך תכונות (למשל צבע, טקסטורה או גובה) לתוצאה נכונה. בלמידת מכונה קלאסית עלינו להגדיר את התכונות (features) בעצמנו: לקבוע מה יוזן למודל, באיזה פורמט ואיך לנרמל או לאזן



איור 1.1: תמונות של חתול וכלב, מתויגות כ־"1" וכ־"0" בהתאמה.

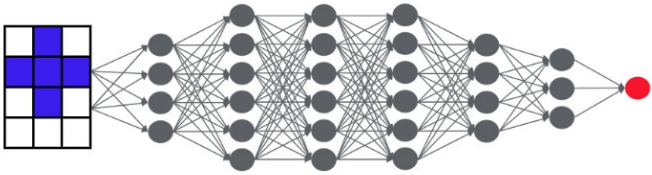


איור 1.2: תהליך האימון: חיזוי ← חישוב טעות ← עדכון משקולות.

את ערכי הקלט.

תפקידה של הלמידה העמוקה

כאן נכנסת לתמונה הלמידה העמוקה. למידה עמוקה אינה טכניקה יחידה, אלא משפחה של שיטות המבוססות על רשתות נוירונים מלאכותיות. כאן המודל מקבל קלט, מעביר אותו דרך שכבות רבות של טרנספורמציה ובונה ייצוגים פנימיים - מבלי שנדרשת הגדרת פיצ'רים ידנית. אם הקלט הוא תמונה למשל, המודל מקבל את כל הפיקסלים הגולמיים ולא רק סט של פיצ'רים שהוגדרו מראש.



איור 1.3: רשת נוירונים עמוקה שמקבלת פיקסלים גולמיים כקלט.

מקרה לדוגמה: זיהוי פיצ'רים

בפרויקט יישומי מסוים, המטרה הייתה לסווג תמונות של מוצרים על בסיס פיצ'רים כמו יוקרה, בהירות וניקיון. הגישה הראשונית, באמצעות למידת מכונה קלאסית, הסתמכה על מאפייני metadata מובנים - מחיר, קטגוריה ודירוגי משתמשים. התוצאה: ביצועים סבירים אך מוגבלים, כי הפיצ'רים הידניים לא היו מספיק עשירים.

במעבר ללמידה עמוקה, אימנו מודלים ישירות על פיקסלי התמונות, ללא פיצ'רים מוגדרים מראש. המודל הצליח ללמוד ייצוגים פנימיים עם תבניות עדינות שלא ניתן היה להגדיר ידנית. זה הדגיש את היתרון המרכזי של למידה עמוקה: היכולת לגלות ולחלץ פיצ'רים רלוונטיים באופן אוטומטי, מבלי להסתמך על הנדסת פיצ'רים.

ייצוג היררכי

למידה עמוקה היא תהליך של למידת ייצוגים היררכית: שכבה אחר שכבה, המידע מעובד ומיוצג ברמות הפשטה הולכות וגדלות - החל מפיקסלים בודדים ועד לאלמנטים מורכבים. זה מה שהופך את הלמידה העמוקה לכל כך עוצמתית, במיוחד עם נתונים לא־מובנים. מאז 2012, עם פריצת הדרך של AlexNet, הלמידה העמוקה חוללה מהפכה בתחומים כמו ראייה ממוחשבת, זיהוי דיבור, תרגום, ניתוח סנטימנט ויצירת תוכן.

מתי לבחור בלמידה עמוקה?

למידה עמוקה היא הבחירה הנכונה כאשר:

- מתמודדים עם נתונים לא־מובנים (תמונות, טקסט, קול).
- יש כמות גדולה מאוד של נתונים זמינים לאימון.
- רוצים שהמערכת תלמד ייצוגים בעצמה במקום להגדיר פיצ'רים באופן ידני.
- המטרה היא לגלות תבניות מורכבות שקשה או בלתי אפשרי להגדירן ידנית.

אתגרים ומגבלות

למרות עוצמתה, ללמידה עמוקה יש אתגרים משמעותיים שיש להביא בחשבון ביישום מעשי: אימון רשתות עמוקות דורש משאבי חישוב גדולים מאוד וזמן אימון ממושך. המודלים סובלים מבעיית "הסברתיות" (interpretability): קשה להסביר למה התקבל חיזוי מסוים, משום שהייצוגים הפנימיים מפוזרים על פני שכבות רבות ומיליוני פרמטרים. מכאן נובע הכינוי "black box" מערכת שבה קשה להתחקות אחרי תהליך קבלת ההחלטות. ובכל זאת, כאשר מאמנים ומוודאים את היציבות של הרשתות, הארכיטקטורות האלו מפיקות תובנות מנתוני ענק ומגלות תבניות שהיו בלתי נגישות בשיטות מסורתיות.

סיכום

למידה עמוקה אינה רק גרסה "מתקדמת יותר" של למידת מכונה - היא שינוי תפיסתי. במקום לתכנן ידנית את הפיצ'רים, אנו בונים מערכות שלומדות לייצג את הנתונים בעצמן באמצעות הפשטות רב־שכבתיות.

ההבדלים המרכזיים:

למידת מכונה קלאסית: הסתמכות על הנדסת פיצ'רים ידנית, ארכיטקטורות פשוטות, נתונים מובנים.

למידה עמוקה: למידת פיצ'רים אוטומטית, שימוש בארכיטקטורות עמוקות, יעילה במיוחד עבור נתונים לא־מובנים.

בשיעורים הבאים נלמד איך לבנות מודלים מסוג זה - החל מהנוירון המלאכותי הבודד ועד לרשתות עמוקות שלמות.

שיעור ראשון בקורס זמין ב-YouTube

שיעור 2

מהי רשת נוירונים

מבוא

בשיעור הקודם בחנו את ההבדל בין למידת מכונה ללמידה עמוקה. ראינו שלמידה עמוקה מאפשרת למודל ללמוד ייצוגים פנימיים בעצמו, ללא הנדסת פיצ'רים ידנית. כעת נעמיק במבנה הפנימי של רשתות נוירונים מלאכותיות, בניסוח המתמטי שלהן ובסיבות שבזכותן הן מסוגלות לייצג פונקציות מורכבות.

הנוירון המלאכותי

בבסיס כל רשת נוירונים נמצא נוירון מלאכותי, או Perceptron. הנוירון מקבל וקטור קלט $\mathbf{x} \in \mathbb{R}^n$, מכפיל כל רכיב במשקולת מתאימה, מסכם את התוצאות, מוסיף bias $b \in \mathbb{R}$ ומעביר את הסכום דרך פונקציית אקטיבציה φ כדי לייצר פלט y:

$$y = \varphi(\mathbf{w} \cdot \mathbf{x} + b) \tag{2.1}$$

כאשר:

- $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$ הוא וקטור הקלט.
- $\mathbf{w} = [w_1, w_2, \ldots, w_n]^\top$ הוא וקטור המשקולות.

- $\mathbf{w} \cdot \mathbf{x}$ היא המכפלה הסקלרית $\sum_{i=1}^{n} w_i x_i$.
- b הוא ערך ה־bias.
- $\varphi(\cdot)$ היא פונקציית אקטיבציה לא־ליניארית.
- y הוא הפלט הסקלרי של הנוירון.

למרות שנוירון בודד הוא פונקציה פשוטה, צירוף של רבים מהם לשכבות מאפשר למודל לייצג פונקציות מורכבות מאוד.

פונקציות אקטיבציה

פונקציית האקטיבציה φ מוסיפה אי־ליניאריות למודל. ללא אי־ליניאריות, גם אם נערום שכבות רבות, נקבל בסופו של דבר טרנספורמציה ליניארית אחת - שאינה מספיקה לבעיות מורכבות. שתי פונקציות נפוצות:

- **ReLU (Rectified Linear Unit)**:

$$\varphi(x) = \max(0, x) \tag{2.2}$$

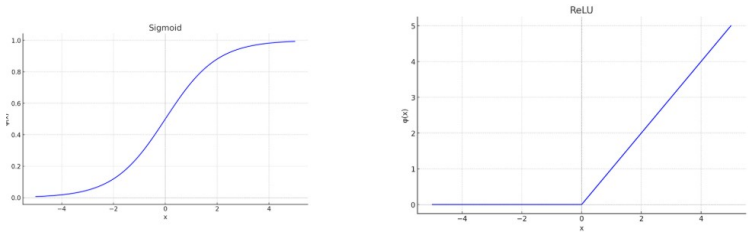
מחזירה 0 עבור ערכים שליליים ומשאירה את הערכים החיוביים כפי שהם. פונקציה זו פשוטה, יעילה חישובית ונפוצה מאוד ברשתות מודרניות.

- **Sigmoid**:

$$\varphi(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

דוחסת את הערכים לטווח $(0, 1)$. שימשה רבות ברשתות מוקדמות, במיוחד במשימות סיווג בינארי, אך כיום השימוש בה מועט בשל בעיית Vanishing Gradient.

טיפ: אם פונקציית המחיר לא יורדת במהלך האימון, אחת הסיבות האפשריות היא בחירה לא מתאימה של פונקציית אקטיבציה - למשל שימוש ב־ *Sigmoid* בשכבות חבויות במקום *ReLU*.



איור 2.1: פונקציות אקטיבציה: Sigmoid (משמאל) ו־ReLU (מימין).

מבנה שכבות ברשת

רשת נוירונים עמוקה מורכבת ממספר רב של שכבות. כל שכבה מקבלת את הפלט של השכבה הקודמת ומחשבת פלט חדש:

$$\mathbf{y}^{(\ell)} = \varphi(\mathbf{W}^{(\ell)}\mathbf{y}^{(\ell-1)} + \mathbf{b}^{(\ell)}) \tag{2.4}$$

כאשר:

- $\mathbf{y}^{(\ell-1)}$ הוא הקלט לשכבה ℓ (כלומר פלט השכבה הקודמת).
- $\mathbf{W}^{(\ell)}$ היא מטריצת המשקולות של שכבה ℓ.
- $\mathbf{b}^{(\ell)}$ הוא וקטור ה־bias של שכבה ℓ.
- φ מופעלת על כל רכיב בנפרד.

הארכיטקטורה הבסיסית כוללת:

- שכבת קלט - מקבלת את הנתונים הגולמיים.
- שכבות חבויות - לומדות ייצוגים פנימיים מורכבים.
- שכבת פלט - מייצרת תחזית בהתאם למשימה.

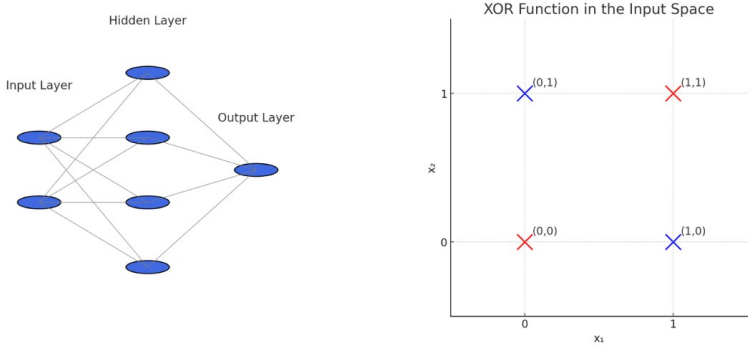
Perceptron רב־שכבתי (MLP)

MLP (Multi-Layer Perceptron) הוא סוג של רשת שבה כל שכבה מחוברת במלואה לשכבה שאחריה.

יתרון מרכזי: MLP מסוגל לייצג בעיות שאינן ניתנות להפרדה ליניארית. דוגמה קלאסית היא פונקציית XOR:

$$y = \begin{cases} 1 & x_1 \neq x_2 \\ 0 & \text{אחרת} \end{cases} \qquad x_1, x_2 \in \{0, 1\} \tag{2.5}$$

פונקציה זו אינה ניתנת להפרדה על ידי קו ישר במרחב הקלט. אך MLP עם שכבה חבויה קטנה אחת בלבד מסוגל ללמוד אותה בהצלחה.



איור 2.2: סיווג XOR: משמאל - רשת נוירונים; מימין - גרף של פונקציית XOR.

משפט הקירוב האוניברסלי

קיימת תוצאה תאורטית חשובה הקרויה The Universal Approximation Theorem. לפיה, גם רשת פשוטה עם שכבה חבויה אחת בלבד - אם יש בה מספיק נוירונים לא־ליניאריים - יכולה לקרב כל פונקציה רציפה (עד לרמת דיוק סופית). בפועל, אימון שכבה אחת ענקית אינו מעשי: הוא דורש הרבה נוירונים ונתקל בבעיות יציבות. לכן משתמשים בארכיטקטורות עמוקות - מספר שכבות קטנות יותר - שמאפשרות ייצוג קומפקטי, גמיש ויעיל בהרבה.

סיכום

רשת נוירונים היא פונקציה פרמטרית הבנויה משכבות של נוירונים מלאכותיים. כל נוירון מחשב צירוף ליניארי משוקלל של הקלט, מוסיף bias ומעביר את התוצאה דרך פונקציית אקטיבציה לא־ליניארית. באמצעות צירוף של שכבות רבות, הרשת מסוגלת ללמוד ייצוגים מורכבים ולהתמודד עם בעיות שלא ניתנות לפתרון באמצעות טרנספורמציה ליניארית בלבד. זהו הצעד הראשון לעבר רשתות עמוקות שמביאות לידי ביטוי את מלוא העוצמה של הלמידה העמוקה.

שיעור 3

פונקציית מחיר, Backpropagation ואופטימיזציה

מבוא

בשיעור הקודם ראינו כיצד מידע עובר קדימה מהקלט לפלט ברשת נוירונים כדי לייצר חיזוי. אבל רשת אינה "נולדת" עם ידע - עליה ללמוד מהנתונים. בשיעור זה נבין כיצד מתבצע תהליך הלמידה, באמצעות שלושה מרכיבים מרכזיים: פונקציית המחיר, Backpropagation ושיטת האופטימיזציה Gradient Descent.

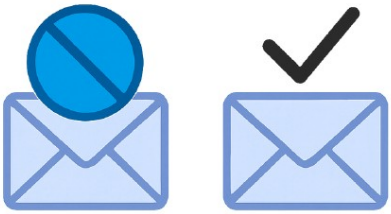
פונקציית מחיר

פונקציית המחיר (Loss Function) מודדת עד כמה החיזוי של המודל רחוק מהתיוג האמיתי. המטרה: לכמת את הפער בין הפלט החזוי לבין התשובה הנכונה.

כי נניח $y \in \{0,1\}$ היא התווית האמיתית (התיוג) ו־$\hat{y} \in (0,1)$ היא ההסתברות שהמודל חזה. פונקציה נפוצה מאוד לסיווג בינארי היא Cross-Entropy:

$$L = -\Big[y \cdot \log(\hat{y}) + (1-y) \cdot \log(1-\hat{y})\Big] \tag{3.1}$$

אם החיזוי קרוב לערך הנכון $L \leftarrow$ קטן. אם החיזוי רחוק או שגוי $L \leftarrow$



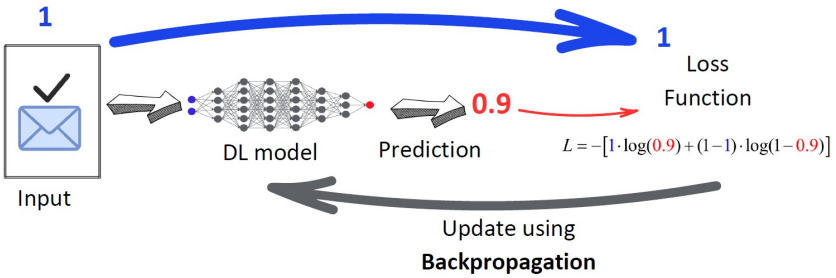
איור 3.1: דוגמה: מייל אחד מסומן כ־Spam והשני לא. המשימה - לחזות נכון.

גדל.

תרשים זרימה מלא של אימון

תהליך האימון מתבצע באופן הבא:

קלט ⟵ חיזוי ⟵ חישוב טעות ⟵ Backpropagation ⟵ עדכון משקולות.



איור 3.2: תרשים זרימת האימון.

Backpropagation

כדי לשפר את המודל, עלינו לעדכן את המשקולות כך שיקטינו את הערך של פונקציית המחיר. Backpropagation הוא האלגוריתם שמאפשר זאת באמצעות חישוב נגזרות ושימוש בכלל השרשרת. הגרדיאנט של פונקציית המחיר ביחס לכל משקל w מוגדר כך:

$$\frac{\partial L}{\partial w} \tag{3.2}$$

באמצעות כלל השרשרת (Chain Rule), מחושבים הגרדיאנטים מהשכבה האחרונה אחורה - דרך שכבות הביניים - עד לקלט. כך ניתן לשייך לכל פרמטר (משקולת) במודל את התרומה שלו לשגיאה ולעדכן אותו בהתאם.

הגרדיאנט - אינטואיציה

נניח שפונקציית המחיר $L(w)$ היא פונקציה חלקה של משקל יחיד w. הנגזרת $\frac{dL}{dw}$ בנקודה מסוימת מתארת את השיפוע באותה הנקודה: כלומר, נוכל לקבוע אם הגדלת w תעלה או תקטין את L.

עדכון המשקולת מתבצע לפי הכלל:

$$w_{k+1} = w_k - \eta \cdot \frac{dL}{dw} \tag{3.3}$$

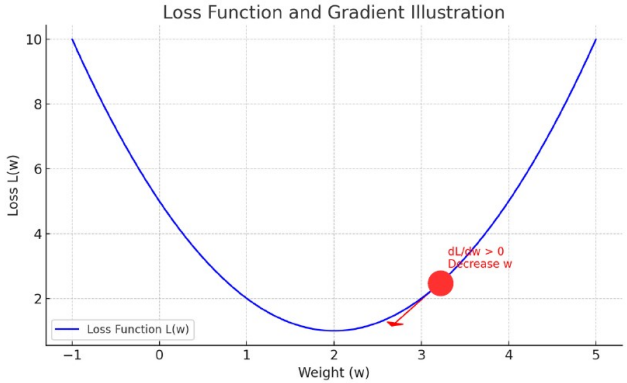
כאשר $\eta > 0$ הוא קצב הלמידה (Learning Rate).

קצב הלמידה

קצב הלמידה η קובע את גודל הצעד בכל עדכון:

- אם η קטן מדי - המודל יתכנס לאט מאוד.

- אם η גדול מדי - המודל עלול להתבדר, כלומר הטעות תגדל במקום לקטון.



איור 3.3: פונקציית המחיר כתלות במשקל. החץ האדום מראה את כיוון הגרדיאנט. העדכון נע בכיוון של הקטנת טעות החיזוי - מזעור פונקציית המחיר.

טיפ: אם במהלך האימון נראה שהטעות "קופצת" למעלה ולמטה ולא מתכנסת לערך יציב, סיבה אפשרית היא שקצב הלמידה גבוה מדי.

כפי שתיאר זאת אחד הסטודנטים: "ניסיתי קצב למידה של 1 והמודל פשוט קפץ הלוך ושוב כמו כדור פינג־פונג."

דוגמה: מודל לינארי מופשט עם משקולת יחידה

נבחן מודל עם משקולת יחידה w וקלט x:

$$\hat{y} = w \cdot x \tag{3.4}$$

פונקציית המחיר - טעות ריבועית ממוצעת (MSE) עבור דוגמה אחת:

$$L = (\hat{y} - y)^2 \tag{3.5}$$

נניח: $x = 2,\ y = 6,\ w = 1$.

החיזוי: $\hat{y} = 1 \cdot 2 = 2$.

הטעות: $L = (2-6)^2 = 16$.

נחשב את הגרדיאנט:

$$\frac{dL}{dw} = 2 \cdot (wx - y) \cdot x \tag{3.6}$$

נציב $w = 1,\ x = 2,\ y = 6$:

$$\frac{dL}{dw} = 2 \cdot (2-6) \cdot 2 = -16$$

עדכון עם $\eta = 0.1$:

$$w_{\text{new}} = w - \eta \cdot \frac{dL}{dw} = 1 - 0.1 \cdot (-16) = 2.6 \tag{3.7}$$

כעת, לאחר עדכון יחיד, המשקל w השתפר משמעותית והתקרב לערך שיקטין את השגיאה.

סיכום

תהליך הלמידה של רשת נוירונים מורכב משלושה שלבים מרכזיים:

1. חישוב החיזוי (Forward Pass).
2. חישוב פונקציית המחיר- מדידת המרחק מהתוצאה הנכונה.
3. Backward Pass - חישוב גרדיאנטים באמצעות Backpropagation ועדכון משקולות בעזרת Gradient Descent.

שלושת המרכיבים האלו הם ליבת תהליך הלמידה ברשתות עמוקות.

שיעור 4

איך תהליך האימון עובד?

עד כה בחנו כיצד רשתות נוירונים בנויות וכיצד הן מייצרות חיזויים. אך הליבה של כל אלגוריתם למידה טמונה ביכולת שלו להשתפר. איך בדיוק רשת לומדת מתוך דאטה? מה מתרחש במהלך האימון וכיצד הפרמטרים - המשקולות וההטיות - מתעדכנים כדי לצמצם את השגיאה?

Batch לעומת Mini-Batch Gradient Descent

בצורתו הקלאסית, הידועה כ־**Batch Gradient Descent**, המודל מעבד את כל דאטהסט האימון במעבר קדימה אחד, מחשב את פונקציית המחיר עבור כל הדוגמאות ומבצע ממוצע כדי לקבל ערך סקלרי יחיד:

$$L_{\text{batch}} = \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, \hat{y}_i) \tag{4.1}$$

כאן N הוא מספר הדוגמאות הכולל באימון, y_i הוא הלייבל האמיתי, $\hat{y}_i$ הוא החיזוי של המודל עבור הקלט x_i ו־ℓ היא פונקציית המחיר לכל דוגמה. הגרדיאנט של פונקציית המחיר ביחס לפרמטרי המודל w מחושב:

$$\nabla_w L_{\text{batch}} = \frac{1}{N}\sum_{i=1}^{N} \nabla_w \ell(y_i, \hat{y}_i) \tag{4.2}$$

לבסוף, המשקולות מתעדכנות באמצעות צעד גרדיאנט יחיד:

$$w_{k+1} = w_k - \eta \cdot \nabla_w L_{\text{batch}} \tag{4.3}$$

שיטה זו מספקת הערכה לגרדיאנט, יציבה ומדויקת, אך אינה יעילה חישובית ודורשת זיכרון רב עבור דאטהסטים גדולים. לכן, היא כמעט ואינה בשימוש בפועל.

במקום זאת, משתמשים בדרך כלל ב־**Mini-Batch Gradient Descent**, שבו הדאטהסט מחולק לקבוצות קטנות יותר בגודל B, הנקראות **Mini-Batches**. עבור כל מיני־באץ', המודל מבצע את השלבים הבאים:

1. מעבר קדימה על כל B הדוגמאות לחישוב החיזויים $\hat{y}_1, \ldots, \hat{y}_B$
2. חישוב פונקציית המחיר עבור כל דוגמה $\ell(y_j, \hat{y}_j)$
3. ממוצע החישובים כדי לקבל את פונקציית המחיר של המיני־באץ':

$$L_{\text{mini}} = \frac{1}{B} \sum_{j=1}^{B} \ell(y_j, \hat{y}_j) \tag{4.4}$$

4. חישוב הגרדיאנט של פונקציית המחיר:

$$\nabla_w L_{\text{mini}} = \frac{1}{B} \sum_{j=1}^{B} \nabla_w \ell(y_j, \hat{y}_j) \tag{4.5}$$

5. עדכון המשקולות:

$$w_{k+1} = w_k - \eta \cdot \nabla_w L_{\text{mini}} \tag{4.6}$$

חשוב: המשקולות מתעדכנות **פעם אחת לכל מיני־באץ'**, רק לאחר חישוב הממוצע והגרדיאנט. גישה זו מאפשרת שימוש יעיל יותר במשאבים חישוביים ומספקת תהליך אופטימיזציה יציב יותר בהשוואה לעדכון על כל דוגמה בודדת.

Epochs, Iterations ו־Updates

כדי להבין כיצד האימון מתקדם לאורך זמן, מגדירים מספר מונחים מקובלים:

- **Epoch** - מעבר מלא אחד על כל דאטהסט האימון.
- **Mini-Batch** - תת־קבוצה מתוך דאטהסט האימון (בגודל B) המשמשת לעדכון משקולות אחד.
- **Iteration** - צעד עדכון משקולות אחד, לאחר עיבוד מיני־באץ' יחיד.

מספר האיטרציות בכל Epoch נקבע על פי:

$$\text{Iterations-per-epoch} = \left\lceil \frac{N}{B} \right\rceil \tag{4.7}$$

דוגמה: עבור $N = 10{,}000$ דוגמאות אימון וגודל מיני־באץ' $B = 32$, מספר האיטרציות בכל Epoch הוא:

$$\left\lceil \frac{10{,}000}{32} \right\rceil = 313 \tag{4.8}$$

משמעות הדבר שהמודל מבצע 313 מעברים קדימה ואחורה בכל Epoch, כל אחד מלווה בעדכון משקולות. רשתות נוירונים מאומנות לרוב על פני עשרות ואף מאות Epochs כדי להגיע להתכנסות.

אם פונקציית המחיר אינה יורדת לאחר מספר *Epochs*, ייתכן שהלמידה הגיעה לרוויה ויש לעצור או לכייל מחדש את הפרמטרים.

Learning Rate

כפי שראינו קודם, בעוד שהגרדיאנט מספק את הכיוון שבו על המודל לשנות את משקולותיו, ה־**Learning Rate** η קובע את גודל הצעד בכיוון זה. אם η קטן מדי - ההתכנסות איטית. אם הוא גדול מדי - תהליך האימון עלול להיות לא יציב, להתנודד או אף להתבדר.

הגרדיאנט מצביע על הדרך - אך ה־*Learning Rate* קובע כמה נלך.

בחירת Learning Rate מתאים היא קריטית לאימון יעיל ויציב.

Optimizers

ה־**Optimizer** הוא האלגוריתם שאחראי על יישום עדכוני הגרדיאנט לפרמטרי המודל. הוא משתמש בגרדיאנטים שחושבו באמצעות Backpropagation, מכפיל אותם ב־Learning Rate η ומבצע עדכון משקולות. הצורה הפשוטה ביותר היא **Gradient Descent**:

$$w_{k+1} = w_k - \eta \cdot \nabla_w L_k \tag{4.9}$$

בפועל, שיטות עבודה מודרניות לאימון עושות שימוש באופטימייזרים מתקדמים יותר כגון **Adam**, **RMSProp** או **Adagrad**, אשר משלבים מנגנונים לייצוב והאצת ההתכנסות.

שיעור 14 יעמיק במבנה המתמטי וביתרונות וחסרונות של אופטימייזרים אלו.

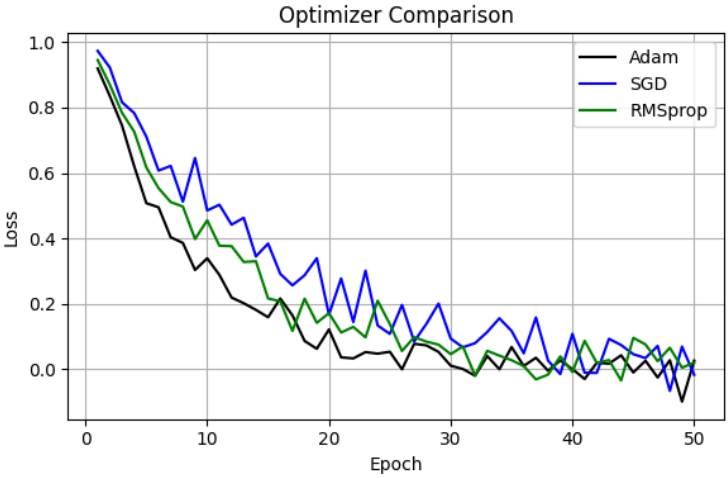
המחשה של דינמיקת האימון

כדי לבחון כיצד היפר־פרמטרים משפיעים על ההתכנסות, ניתן לדמות אימון מודל על דאטהסט סינתטי תוך שינוי האופטימייזר, ה־Learning Rate וגודל ה־Batch.

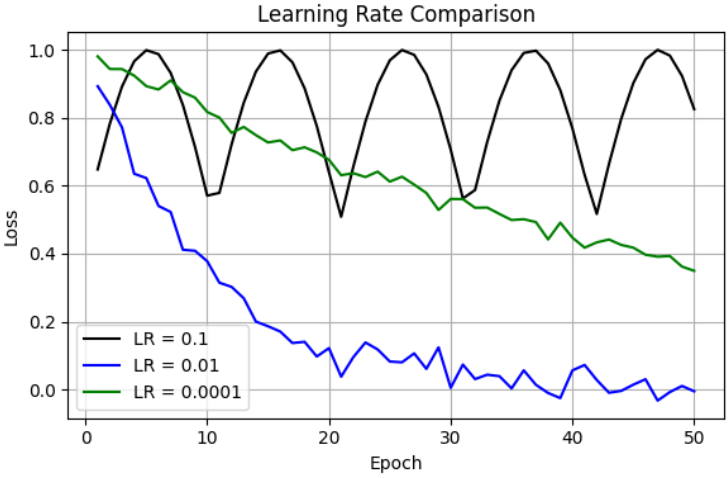
התנהגות Optimizer: Adam (שחור) מתכנס במהירות ובצורה חלקה. SGD (כחול) מציג חוסר יציבות. RMSProp (ירוק) יציב יותר מ־SGD אך פחות יעיל מ־Adam.

התנהגות Learning Rate: קצב למידה גבוה (שחור) מוביל לעדכונים רועשים או מתבדרים. קצב נמוך (ירוק) מאט את ההתכנסות. ערך בינוני (כחול) מספק איזון בין יציבות למהירות.

התנהגות Batch Size: באצ'ים קטנים (שחור) מייצרים עדכונים מהירים אך רועשים. באצ'ים בינוניים (כחול) מספקים איזון. באצ'ים גדולים (ירוק) מובילים להתכנסות איטית אך חלקה יותר.



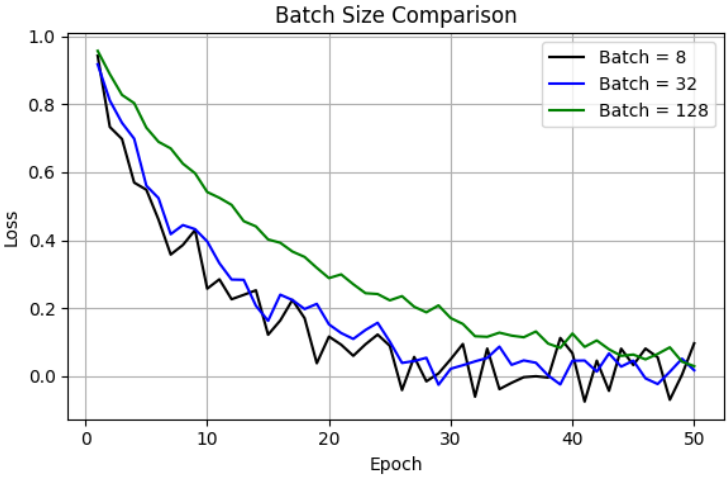
איור 4.1: השוואת Optimizers - Loss לעומת Epoch. שחור: Adam, כחול: SGD, ירוק: RMSProp.



איור 4.2: השוואת Learning Rate. שחור: גבוה, כחול: בינוני, ירוק: נמוך.

סיכום

דינמיקת האימון נקבעת על ידי האינטראקציה בין מספר רכיבים מרכזיים. **גודל ה־Batch** מכתיב כמה דוגמאות מעובדות יחד לצורך חישוב עדכון



איור 4.3: השוואת Batch Size. שחור: קטן, כחול: בינוני, ירוק: גדול.

גרדיאנט ומשפיע על היציבות והוריאנס של תהליך האופטימיזציה. **מספר ה־Epochs** קובע כמה פעמים המודל עובר על כל דאטהסט האימון ומשפיע על החשיפה הכוללת לדאטה. **ה־Learning Rate** η מגדיר את גודל צעד העדכון במרחב הפרמטרים ומאזן בין מהירות ההתכנסות לבין סיכון של פספוס. לבסוף, ה־**Optimizer** מגדיר את חוק העדכון בפועל ולעיתים משלב מנגנונים כמו Momentum או התאמות אדפטיביות לשיפור יעילות הלמידה.

אימון אפקטיבי תלוי בבחירה נכונה של ערכים לכל אחד מהרכיבים הללו. גם טעויות קטנות עלולות לגרום לעדכונים לא יציבים, להתכנסות איטית, או לכישלון בלמידה.

בשיעור הבא נעבור מאופטימיזציה להערכה ונבחן כיצד לקבוע אם המודל אכן למד ומסוגל להכליל על דאטה שלא ראה קודם.

שיעור 5

מדדי הערכת ביצועים

מבוא

בשיעורים הקודמים בנינו רשת נוירונים, אימנו אותה על דאטה אמיתי, עדכנו משקולות באמצעות גרדיאנטים ובחרנו אופטימייזר. כעת מגיעה השאלה הקריטית: איך נדע אם המודל שלנו באמת טוב?

המטרה של Machine Learning אינה לשנן את דאטה האימון, אלא להכליל - כלומר, להצליח בביצועים טובים גם על דאטה חדש שלא נראה קודם. יכולת זו נקראת **יכולת ההכללה** של המודל והיא המדד המרכזי של כל מערכת חיזוי.

פיצול דאטהסט

לפני שמתחיל כל אימון - אפילו לפני בחירת הארכיטקטורה של המודל - הדאטהסט חייב להיות מחולק לתת־קבוצות נפרדות. חלוקה זו חיונית כדי לוודא שביצועי המודל משקפים למידה אמיתית ולא שינון. ה־**Training Set** משמש להתאמת המודל: חישוב הגרדיאנטים מתבצע עליו והפרמטרים מתעדכנים בהתאם. ה־**Validation Set** נשמר בצד במהלך האימון ומשמש לניטור ביצועים, כיוון היפר־פרמטרים וזיהוי סימני Overfitting. לבסוף, ה־**Test Set** נשמר אך ורק להערכת ההכללה הסופית - אין להשתמש בו בשום שלב של בחירת מודל או כיול.

אסטרטגיה נפוצה מחלקת בערך 70% מהדאטה לאימון, 15% לולידציה

ו־15% לבדיקה. עם זאת, היחסים עשויים להשתנות בהתאם לגודל ולאופי הדאטהסט. העיקרון המרכזי הוא שה־Test Set חייב להישאר בלתי נגיש עד הסוף, כדי לספק הערכה אובייקטיבית לביצועי אמת.

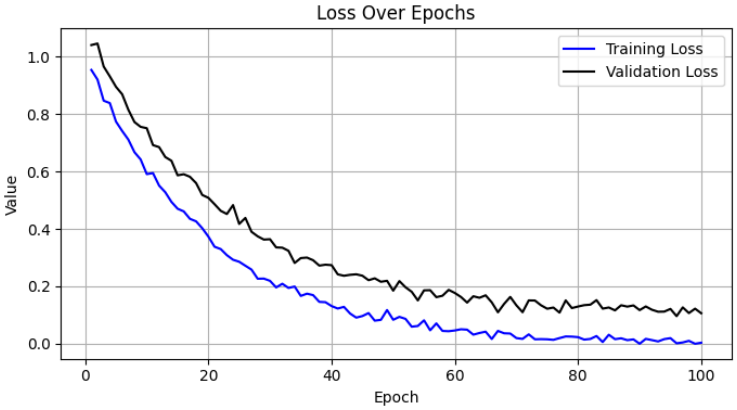
מעקב אחרי Loss ו־Accuracy

במשימות סיווג עוקבים אחרי שני מדדים מרכזיים לאורך האימון: **Loss** ו־**Accuracy**. מחושבים בנפרד על דאטהסט האימון ועל דאטהסט הוולידציה.

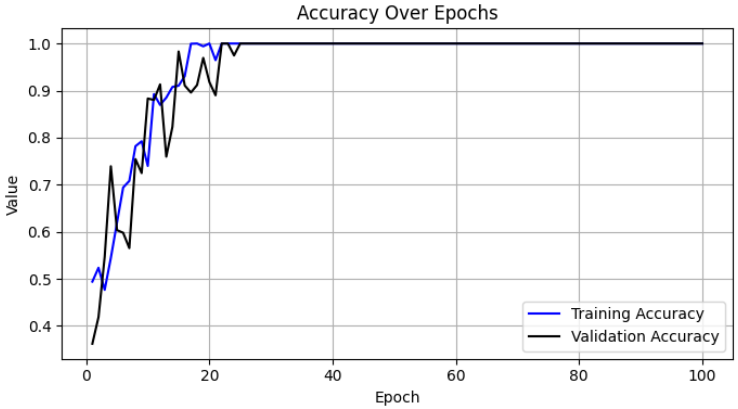
פונקציית ה־Loss מספקת אות רציף וגזיר המראה עד כמה רחוקות תחזיות המודל מהלייבלים האמיתיים. היא רגישה לביטחון התחזיות ומענישה יותר על שגיאות גדולות. לעומת זאת, Accuracy הוא מדד בדיד, הבודק את שיעור התחזיות הנכונות:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{5.1}$$

שימו לב: ערך נמוך של Loss לא בהכרח מעיד על Accuracy גבוה - ה־Loss בוחן את ביטחון התחזית, בעוד Accuracy מתייחס רק לנכונות התיוג.



איור 5.1: מחיר (Loss) באימון ובוולידציה לאורך Epochs: כחול - אימון, שחור - ולידציה.



איור 5.2: דיוק (Accuracy) באימון ובוולידציה לאורך Epochs: כחול - אימון, שחור - ולידציה.

Confusion Matrix ודוגמה

כדי להבין טוב יותר את התנהגות המודל בסיווג בינארי, במיוחד בתחומים רגישים כמו בריאות, משתמשים בכלי בשם **Confusion Matrix**. הוא משווה תחזיות עם לייבלים אמיתיים ומחלק את התוצאות לארבעה מצבים:

Confusion Matrix

Actual Negative	**Actual Positive**	
False Positive (FP)	True Positive (TP)	**Predicted Positive**
True Negative (TN)	False Negative (FN)	**Predicted Negative**

הדיוק (Accuracy) במונחי מרכיבי הטבלה:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.2}$$

דוגמה: זיהוי מחלה

נניח סיווג בינארי לגילוי מחלה נדירה שבה רק אחד מכל 100 אנשים חולה. אם המודל ינבא "בריא" לכולם, הוא ישיג 99% Accuracy - מספרית גבוה, אך מעשית קטסטרופה. מטריצת הבלבול חושפת מקרים מטעים כאלה.

Healthy Predicted:	**Sick Predicted:**	
(Missed) FN	(Correct) TP	**Sick Actually**
(Correct) TN	(False Alarm) FP	**Healthy Actually**

לכל רבעון במטריצה יש השלכות בעולם האמיתי:

TP המחלה זוהתה נכון.

FN חולה הוחמץ - כשל קריטי.

FP בריא זוהה כחולה - יוצר חרדה ועלות מיותרת.

TN בריא סווג נכון.

בהגדרות כאלה, ה־"חיובי" מתייחס למצב שאותו רוצים לגלות - כאן, חולי. כל המדדים הבאים (Precision, Recall, F1) מחושבים ביחס למחלקה החיובית.

Precision, Recall ו־F1 Score

מתוך המטריצה מגדירים מדדים אינפורמטיביים יותר:

Precision - איזה חלק מהתחזיות החיוביות נכונות?

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.3}$$

Recall (Sensitivity) - איזה חלק מהמקרים החיוביים זוהו?

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.4}$$

F1 Score - הממוצע ההרמוני של Precision ו־Recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.5}$$

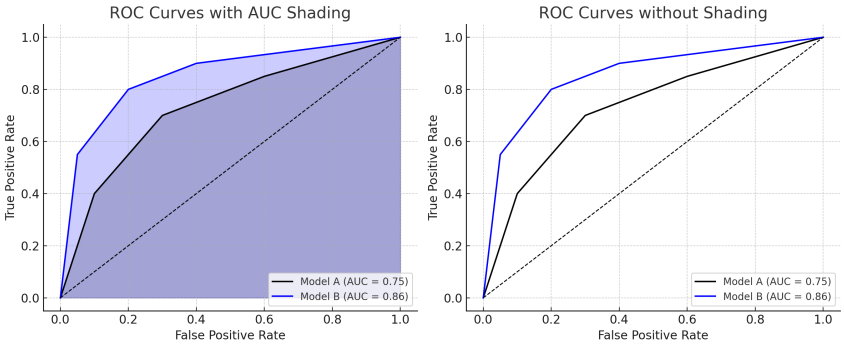
מדדים אלה חשובים במיוחד בדאטהסטים לא מאוזנים. Precision גבוה עם Recall נמוך מעיד על מודל שמרני; Recall גבוה עם Precision נמוך מרמז על הרבה התראות שווא.

ROC Curve ו־AUC

כאשר מודלים מחזירים הסתברויות, מגדירים סף החלטה (בדרך כלל 0.5) כדי להמיר לסיווג בינארי. שינוי הסף משפיע על רגישות וספציפיות.

עקומת ה־ROC מתארת את True Positive Rate (Recall) מול False Positive Rate עבור ספים שונים. השטח שמתחת לעקומה - ה־AUC - מסכם את יכולת ההבחנה של המודל בין מחלקות:

- AUC = 1.0 : סיווג מושלם
- AUC = 0.5 : לא טוב מניחוש אקראי
- AUC < 0.5 : תחזיות הפוכות



איור 5.3: עקומת ROC עם AUC (שטח מוצלל): מודל A - שחור, AUC=0.75; מודל B - כחול, AUC=0.86.

מדדי רגרסיה

כאשר המודל מחזיר ערך רציף ולא מחלקה, משתמשים במדדי **Regression**. אלו מודדים את הפער בין ערכים חזויים $\hat{y}_i$ לערכים אמיתיים y_i.

Mean Absolute Error (MAE)

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{5.6}$$

MAE מודד את גודל השגיאה הממוצע, מתייחס באותו אופן לכל סטייה. הוא חסין למדידות חריגות ובעל יחידות זהות למשתנה החזוי.

Mean Squared Error (MSE)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.7}$$

MSE מדגיש שגיאות גדולות בגלל העלאה בריבוע. נפוץ כשמשתמשים בו כפונקציית מחיר כי הוא רציף וגזיר, אך רגיש יותר לחריגים.

מקדם ההסבר (R^2)

ה־**Coefficient of Determination**, המסומן R^2, הוא מדד נפוץ להערכת מודלי רגרסיה. הוא מודד עד כמה החיזויים $\hat{y}_i$ קרובים לערכים האמיתיים y_i, בהשוואה לבייסליין פשוט - ניבוי תמידי של הממוצע.

פורמלית:

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} \tag{5.8}$$

כאשר SS_{res} הוא סכום ריבועי השגיאות בין חיזויים לערכים אמיתיים ו־SS_{tot} הוא סך הווריאציה בערכים האמיתיים סביב הממוצע $\bar{y}$:

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

$$SS_{\text{res}} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \quad SS_{\text{tot}} = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

ערך $R^2 = 1$ מצביע על התאמה מושלמת. ערך $R^2 = 0$ מצביע על חיזוי גרוע כמו הממוצע. אם $R^2 < 0$, המודל גרוע מהבייסליין.

מדדים נוספים לסיווג

Balanced Accuracy

מדד זה מתחשב בדאטהסטים לא מאוזנים על ידי ממוצע ה־Recall בין מחלקות:

$$\text{Accuracy Balanced} = \frac{1}{2}\left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\right) \tag{5.9}$$

כך כל מחלקה משפיעה באופן שווה, בלי קשר לשכיחות.

Matthews Correlation Coefficient (MCC)

$$\text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{5.10}$$

ה־MCC הוא מדד מאוזן גם מול דאטה לא מאוזן. מתחשב בכל ארבע התוצאות ומתנהג כמדד מתאם: $+1$ = תחזית מושלמת, 0 = ניחוש אקראי, -1 = חוסר התאמה מוחלט.

סיכום

הערכת ביצועי מודל דורשת יותר ממספר יחיד. בסיווג, ה־Accuracy הוא נקודת פתיחה, אך עומק נוסף מגיע מ־Precision, Recall, F1 Score ו־AUC. ברגרסיה, MAE, MSE ו־R^2 מודדים את איכות התחזיות המספריות. מדדים כמו Balanced Accuracy ו־MCC מספקים רובסטיות בסביבות לא מאוזנות. מעל הכל, הערכה חייבת להתבצע על Test Set שמור כדי לספק הערכת הכללה אמינה.

שיעור 6

Overfitting ו־Regularization

מבוא

בשלבים המוקדמים של אימון רשת נוירונים, הכול עשוי להיראות תקין: ה־Training Loss יורד בהדרגה וה־Training Accuracy משתפר מאפוק לאפוק. כאשר מוצג דאטה חדש ולא נראה, ביצועי המודל עלולים לקרוס בפתאומיות. תופעה זו אינה באג - זוהי בעיה יסודית בלמידה עמוקה, הידועה בשם **Overfitting** (התאמת יתר).

Overfitting מתרחש כאשר המודל מתאים עצמו יותר מדי לדאטהסט האימון, קולט לא רק את הדפוסים הבסיסיים אלא גם רעשים, חריגים וקורלציות מקריות. כתוצאה מכך, המודל מצליח על הדאטה שראה, אך נכשל בהכללה על דוגמאות חדשות. לעומת זאת, **Underfitting** (תת התאמה), מתאר מצב שבו המודל אינו מצליח לקלוט את הדפוסים בדאטהסט האימון, לרוב בשל קיבולת לא מספקת או אימון לא מספק.

גם Overfitting וגם Underfitting פוגעים ביכולת ההכללה - שהיא המטרה המרכזית.

דוגמה

דמיינו תלמיד ששינן תשובות ממבחן בשנה שעברה מבלי להבין את החומר. כאשר מוצג לו מבחן חדש עם שאלות מעט שונות, הוא אינו מסוגל לענות

נכון. האנלוגיה פשוטה: שינון איננו למידה.

באופן דומה, רשת נוירונים עשויה להמשיך להקטין את Training Loss, בעוד Validation Loss מתחיל לעלות - סימן קלאסי ל־Overfitting. בתחילה המודל משתפר גם על אימון וגם על ולידציה, אך בשלב מסוים הוא מתחיל לשנן במקום ללמוד.

התנהגות ה־Loss ככלי אבחון

במהלך האימון עוקבים אחרי ה־Loss גם על דאטהסט האימון וגם על הוולידציה. נסמן $L_{\text{train}}(t)$ ו־$L_{\text{val}}(t)$ כמחיר באימון ובולידציה, בהתאמה, באפוק t.

בשלבים המוקדמים:

$$L_{\text{train}}(t) \downarrow, \quad L_{\text{val}}(t) \downarrow$$

בנקודת תחילת Overfitting:

$$L_{\text{train}}(t) \downarrow, \quad L_{\text{val}}(t) \uparrow$$

הסטייה הזו היא סימן קריטי לכך שהמודל כבר לא לומד מבנה משמעותי, אלא משנן את דאטהסט האימון.

מדוע מתרחש Overfitting

רשתות נוירונים מודרניות מכילות לעיתים מיליוני פרמטרים. עם מספיק קיבולת, המודל יכול תמיד להשיג Training Loss כמעט אפסי - אפילו על דאטה עם תוויות אקראיות. אך פתרון כזה לומד רעש במקום מבנה בסיסי.

דיוק מושלם על דאטהסט האימון הוא לרוב דגל אדום, לא סיבה לחגיגה. הוא מצביע על עודף פרמטרים ועל היעדר Inductive Bias.

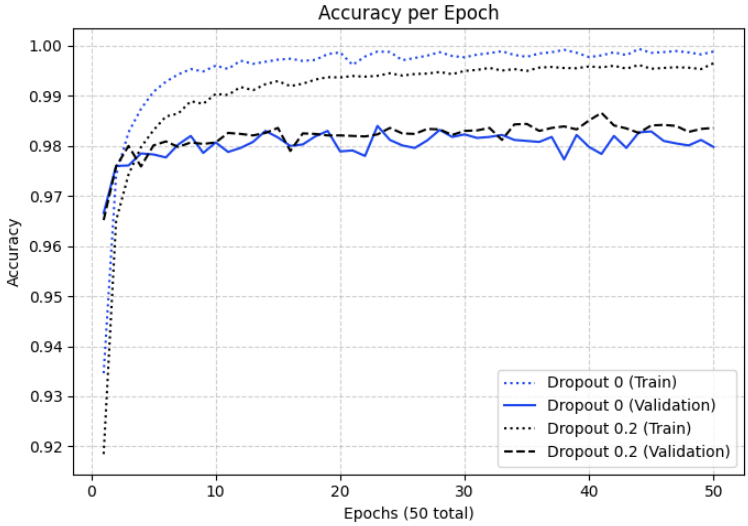
שלוש טכניקות ליבה למניעת Overfitting

1. Dropout

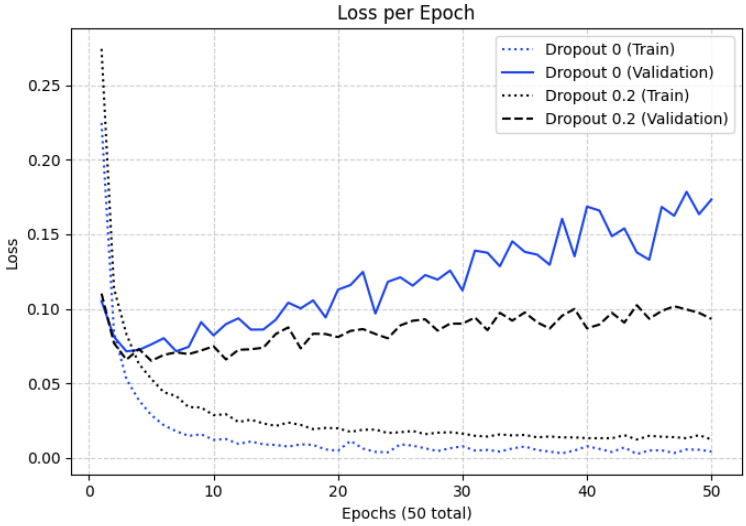
Dropout היא טכניקת רגולריזציה פשוטה אך עוצמתית. במהלך האימון, כל נוירון חבוי נופל - כלומר מושבת זמנית - בהסתברות p. כך, בכל מעבר קדימה מתאמנות תת־רשתות שונות.

פורמלית, אם $h^{(l)}$ הן האקטיבציות בשכבה l, אזי Dropout מגדיר מסיכה בינארית $m^{(l)} \sim \text{Bernoulli}(1-p)$: $\tilde{h}^{(l)} = m^{(l)} \cdot h^{(l)}$. בשלב Inference, כל הנוירונים פעילים והמשקולות מותאמות בקנה מידה כדי לפצות על היעדר הדרופ־אאוט.

Dropout מונע הסתגלות־יתר של נוירונים ומעודד פיתוח ייצוגים רובסטיים וחופפים. התוצאה - שיפור בהכללה.



איור 6.1: דיוק לאורך Epochs. כחול: ללא דרופ־אאוט. מקווקו שחור: דרופ־אאוט עם $p = 0.2$.



איור 6.2: ולידציה Loss לאורך Epochs. כחול: ללא דרופ־אאוט. מקווקו שחור: דרופ־אאוט עם $p = 0.2$.

2. Early Stopping

Early Stopping עוקב אחרי Validation Loss בזמן האימון. אם לא נצפה שיפור לאורך מספר מוגדר של אפוקים (נקרא Patience), האימון נעצר מוקדם.

אסטרטגיה פשוטה זו מונעת מהמודל להמשיך ולמזער את Training Loss על חשבון הכללה. היא שימושית במיוחד כשמספר האפוקים האופטימלי אינו ידוע מראש.

3. Weight Decay (L2 Regularization)

L2 Regularization, המכונה גם *Weight Decay*, מענישה משקולות גדולות באמצעות הוספת איבר ריבועי לפונקציית המחיר:

$$L_{\text{reg}} = L + \lambda \sum_i w_i^2 \tag{6.1}$$

כאן L הוא ה־Loss המקורי (למשל, Cross-Entropy), w_i הוא המשקל

ה־i ו־λ הוא מקדם הרגולריזציה. האינטואיציה: משקולות גדולות מצביעות על תלות חזקה בפיצ'רים מסוימים. ענישת משקולות כאלה מעודדת את הרשת לפזר את החשיבות בצורה שוויונית יותר ובכך מובילה ללמידה יציבה ומוכללת.

טכניקה	מנגנון	מטרה
Dropout	משבית אקראית נוירונים במהלך האימון	מונע התאמת־יתר, מעודד חפיפה ורובסטיות
Early Stopping	עוצר אימון כש־Validation Loss מפסיק להשתפר	מונע אופטימיזציה עודפת מעבר לנקודת ההכללה
L2 Regularization (Weight Decay)	מוסיף איבר ענישה $\lambda \sum w_i^2$ לפונקציית המחיר	מרכך משקולות גדולות, מעודד פתרונות חלקים

טבלה 6.1: השוואת טכניקות רגולריזציה נפוצות.

מתי ליישם רגולריזציה?

בתרחישים רבים בעולם האמיתי, במיוחד כאשר דאטהסט הוולידציה קטן או רועש, קשה לזהות בוודאות Overfitting. לכן, לרוב מומלץ ליישם טכניקות רגולריזציה כברירת מחדל - כאמצעי מניעה, לא רק כתגובה לכישלון.

סיכום

בשיעור זה בחנו את אחד האתגרים החשובים ביותר בלמידה עמוקה: Overfitting. הגדרנו אותו, זיהינו אותו באמצעות עקומות Loss והצגנו שלושה כלים עוצמתיים להתמודדות: Dropout, Early Stopping ו־Weight Decay.

טכניקות אלו אינן רק מגבילות את המודל, אלא מנחות אותו ללמוד דפוסים משמעותיים במקום לשנן רעש. במודול הבא נעמיק בטכניקות אימון מתקדמות וארכיטקטורות שמאפשרות רשתות עמוקות ומבטאות יותר - מבלי לוותר על יכולת ההכללה.

שיעור 7

מדוע אנחנו צריכים קונבולוציה?

מבוא

בשיעורים הקודמים בחנו כיצד רשתות נוירונים לומדות מדאטה מובנה. בשיעור זה נתמקד בתמונות - תחום עם מבנה מרחבי חזק.

רשתות (MLPs) מתעלמות ממבנה זה ודורשות מספר עצום של פרמטרים (משקולות). רשתות קונבולוציה (CNNs) מציעות פתרון יעיל וסקלאבילי יותר. אך מדוע בכלל דרושה קונבולוציה?

שכבות Fully Connected והמגבלות שלהן

נבחן תמונת RGB קטנה בגודל 32×32 פיקסלים. מספר הפיצ'רים בקלט הוא:

$$3 \times 32 \times 32 = 3{,}072 \tag{7.1}$$

אם וקטור זה משוטח ומחובר לשכבה צפופה של 100 נוירונים, מספר המשקולות בשכבה הראשונה בלבד יהיה:

$$W_{Total} = 3072 \times 100 = 307{,}200 \tag{7.2}$$

חיבור מלא זה מאבד מידע מרחבי - הרשת אינה יודעת שפיקסלים סמוכים קשורים זה לזה. ככל שנוסיף שכבות נוספות, מספר הפרמטרים

יגדל במהירות ועלול להוביל ל־Overfitting.

טבלה 7.1: מספר פרמטרים ומודעות מרחבית

ארכיטקטורה	מספר פרמטרים	מידע מרחבי
MLP (100 נוירונים)	307,200	×
CNN (32 פילטרים של 3×3)	864	✓

מדוע קונבולוציה עובדת - Inductive Bias

במדעי ה־Machine Learning ידוע משפט בשם No Free Lunch. הוא קובע שאם מחשבים ממוצע ביצועי אלגוריתם על פני כל הפונקציות האפשריות ליצירת דאטה, אף שיטה אינה טובה מאחרת. הסיבה היחידה לכך ש־CNNs עדיפה על MLPs בתמונות היא ההנחה על **לוקאליות ואינווריאנטיות להזזה**. באופן דומה, ארכיטקטורות אחרות (כמו Transformers) עובדות טוב עם טקסט בזכות היכולת לדגם יחסים בין כל זוג טוקנים.

בפועל, המשמעות היא שאין מודל אחד שהוא "הטוב ביותר" בכל מצב, אלא מודלים שמתאימים למבנה של הדאטה שלך. ארכיטקטורות קונבולוציה מצליחות במשימות ראייה בדיוק מפני שהן מקודדות הנחות על האופן שבו תבניות ויזואליות מתנהגות.

אופרטור הקונבולוציה

קונבולוציה מפעילה פילטר קטן (שאיבריו נלמדים) על אזורים מקומיים בתמונה. נניח $I \in \mathbb{R}^{H \times W}$ היא תמונת קלט בגווני אפור ו־$K \in \mathbb{R}^{k \times k}$ הוא Kernel (פילטר). הקונבולוציה הדיסקרטית הדו־ממדית מוגדרת כך:

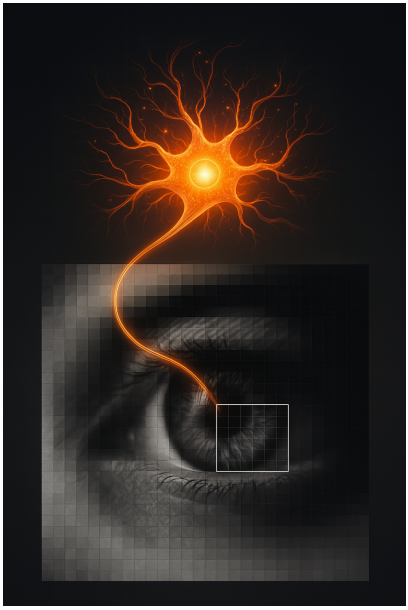
$$F(i,j) = \sum_{u=1}^{k} \sum_{v=1}^{k} K(u,v) \cdot I(i+u-1, j+v-1) \tag{7.3}$$

כאשר:

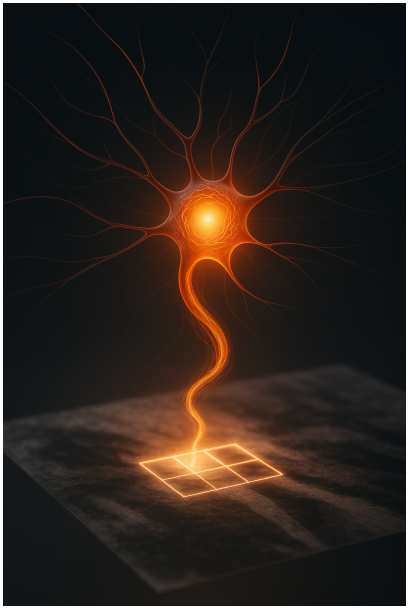
- $F(i,j)$ הוא ערך המיפוי הפיצ'רי ביציאה במיקום (i,j).

- $K(u,v)$ הוא משקל הפילטר במיקום (u,v).
- $I(i+u-1, j+v-1)$ הוא ערך הפיקסל בקלט עבור חלון הפילטר.

פעולה זו חוזרת על כל המיקומים המרחביים, תוך שימוש באותו פילטר - מה שמקטין מאוד את מספר הפרמטרים הנלמדים.



איור 7.1: פילטר המופעל על אזור העין - גילוי תבנית מקומית.



איור 7.2: שימוש חוזר בפילטר - אינווריאנטיות להזזה.

אינטואיציה של קונבולוציה

קונבולוציה מחלצת תבניות לשימוש חוזר מאזורים מקומיים. למשל, פילטר עשוי לזהות "עין" או "פינה" שמופיעים בכל מקום בתמונה. בדומה למוח האנושי שאינו משנן אובייקטים שלמים אלא לומד את רכיביהם (קווים, קשתות, עיניים), CNNs מתמחות בזיהוי תבניות שאינן תלויות במיקום מרחבי.

מאפיינים

אחד היתרונות המרכזיים של קונבולוציה הוא **שיתוף פרמטרים** - אותה קבוצת משקולות מופעלת על מיקומים שונים במרחב. כך מספר הפרמטרים הנלמדים קטן מאוד והיעילות החישובית משתפרת. בנוסף, מודלים קונבולוציוניים רובסטיים מטבעם להזזה - הם יכולים לזהות עצמים ללא תלות במיקומם בתמונה - מה שמוביל לשיפור בהכללה וביציבות. למרות שקונבולוציה עשויה להקטין מימדים מרחביים, התנהגותה נשלטת גם על ידי:

- **Stride** (s): מספר הפיקסלים שהפילטר זז בכל צעד.
- **Padding** (p): מספר הפיקסלים המוספים לקצוות כדי לשמר את גודל הפלט.

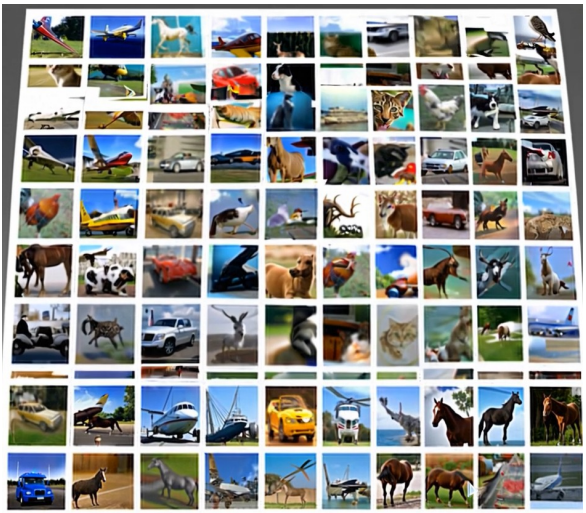
הגדרות אלו יינתנו פורמלית בשיעור הבא.

דאטהסט CIFAR-10

כדי להשוות בין MLPs ו־CNNs, נשתמש בבנצ'מרק **CIFAR-10** - דאטהסט סטנדרטי לסיווג תמונות. הוא כולל 60,000 תמונות צבע בגודל $32 \times 32 \times 3$, בעשר מחלקות כגון מטוס, כלב ורכב.

הסיבות לשימוש הנפוץ ב־CIFAR-10:

- קטן מספיק לניסויים מהירים.
- כולל שונות אמיתית מעולם האמיתי בעשר מחלקות מגוונות.
- CNNs מתקדמים מגיעים לדיוק מעל 90%.



איור 7.3: דוגמאות מתוך CIFAR-10.

מקרה לדוגמה: MLP לעומת CNN על CIFAR-10

אימנו שני מודלים על CIFAR-10 במשך 30 אפוקים:

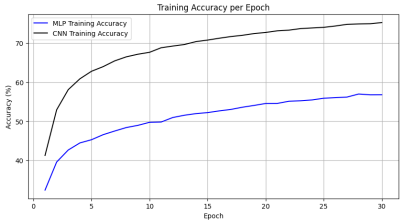
1. MLP מלא עם 2 שכבות צפופות ואקטיבציית ReLU.
2. CNN עם שתי שכבות קונבולוציה ואקטיבציית ReLU.

תוצאות:

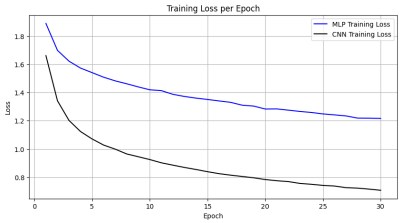
- MLP הגיע רק לדיוק של ∼50%.
- CNN הגיע לכ־68%, טוב בהרבה.

סיכום

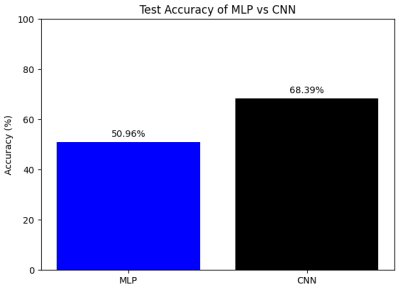
קונבולוציה מציגה Inductive Bias חזק המותאם לדאטה ויזואלי. בניגוד לרשתות MLPs שמתעלמות מהמבנה המרחבי, CNNs שומרות על לוקאליות של פיצ'רים על ידי עיבוד מידע באזורים מקומיים חופפים. הדבר מאפשר להן ללמוד תבניות מרחביות משמעותיות כגון קצוות, טקסטורות וצורות. בזכות מנגנון שיתוף המשקולות, CNNs דורשות הרבה פחות פרמטרים -



איור 7.4: דיוק באימון: CNN (שחור) מגיע לדיוק גבוה מ־MLP (כחול), מהר יותר.



איור 7.5: מחיר באימון: CNN (שחור) מגיע לשגיאה נמוכה יותר מ־MLP (כחול), מהר יותר.



איור 7.6: מחיר באימון - CNN מתכנס מהר יותר ומגיע לשגיאה נמוכה יותר.

מה שמאפשר שיפור בהכללה וגם אימון יעיל יותר. יתרון נוסף הוא יכולתן לזהות תבניות בכל מקום בתמונה, מה שמספק רובסטיות להזזה. תכונות אלו הופכות את CNNs ליעילות וסקלאביליות מאוד עבור דאטה ויזואלי מהעולם האמיתי, גם כאשר ממדי הקלט גדלים.

בשיעור הבא ננסח פורמלית את ההגדרות המתמטיות של Stride, Padding ו־Pooling ונבנה ארכיטקטורות קונבולוציה מלאות למשימות שונות.

שיעור 8

איך פועלת רשת CNN?

מבוא

כעת, לאחר שהבנו מדוע קונבולוציה חיונית, נפנה למבנה הפנימי של רשת קונבולוציה (CNN). בשיעור זה נגדיר את רכיבי שכבת הקונבולוציה, נסביר כיצד פילטרים גולשים על פני התמונה ונציג מנגנונים קריטיים כגון Stride, Padding ו־Pooling.

CNNs פועלות על ידי חילוץ פיצ'רים מקומיים באמצעות פילטרים נלמדים. פילטרים אלה מבצעים קונבולוציה על פני התמונה ומייצרים Feature Maps - כל אחד מהם מדגיש תבנית אחרת כגון קצוות, פינות או טקסטורות. כאשר שכבות קונבולוציה נערמות באופן היררכי, הן מאפשרות הבנה סמנטית עמוקה של דאטה חזותי.

שכבת קונבולוציה: פילטרים וחלון הזהה

כל שכבת קונבולוציה מורכבת ממספר פילטרים (או Kernels). כל פילטר הוא מטריצה נלמדת של משקולות, בדרך כלל בגודל $k \times k$, למשל 3×3.

בכל מיקום מרחבי, הפילטר מבצע כפל איבר־איבר עם הקלט המתאים, מסכם את התוצאה ומזין אותה למפת הפיצ'רים ביציאה. פעולה זו נקראת Sliding Window Convolution.

פורמלית, נניח $I \in \mathbb{R}^{H \times W}$ היא תמונת גווני אפור ו־$K \in \mathbb{R}^{k \times k}$ הוא

פילטר קונבולוציה. הפלט $F \in \mathbb{R}^{H_o \times W_o}$ מוגדר כך:

$$F(i,j) = \sum_{u=1}^{k} \sum_{v=1}^{k} K(u,v) \cdot I(i+u-1, j+v-1) \tag{8.1}$$

כאן $F(i,j)$ הוא ערך הפלט במיקום (i,j), $K(u,v)$ הוא מקדם הפילטר במיקום (u,v) ו־$I(i+u-1, j+v-1)$ הוא פיקסל הקלט בחלון הפילטר.

הפעלת M פילטרים כאלה מייצרת M Feature Maps - אחד לכל תבנית שהרשת למדה לזהות.

Padding והשפעות קצה

ללא Padding, הפילטר אינו יכול לכסות את אזורי הקצה של התמונה במלואם. כתוצאה מכך, כל קונבולוציה מקטינה את המימדים המרחביים של מפת הפיצ'רים. כדי לשמר את הממדים, מוסיפים **Zero-Padding** - הוספת שורות ועמודות של אפסים סביב הקלט.

$$x = \begin{bmatrix} ? & ? & & \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 0 \\ 0 & 5 & 6 & 7 & 8 & 0 \\ 0 & 9 & 10 & 11 & 12 & 0 \\ 0 & 13 & 14 & 15 & 16 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

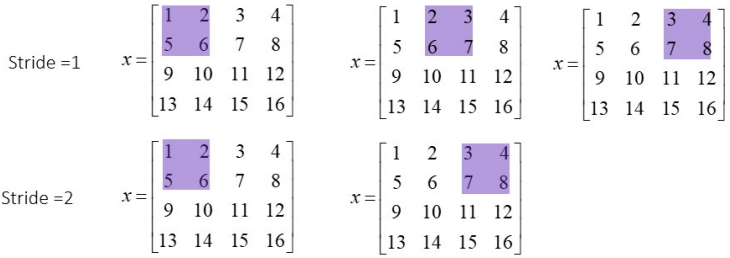
איור 8.1: Zero-Padding : קצוות נוספים מאפשרים כיסוי מלא של מימדי הקלט.

Padding חשוב במיוחד ברשתות עמוקות, שבהן שכבות קונבולוציה רבות עלולות להקטין את הרזולוציה המרחבית יתר על המידה.

Stride והשפעתו

ה־**Stride** s קובע בכמה פיקסלים הפילטר זז בכל צעד. ערך 1 אומר שהפילטר זז פיקסל אחד בכל פעם - ויוצר פלט ברזולוציה גבוהה. ערך 2 מדלג על

מיקומים חלופיים, מצמצם את ממדי הפלט ומייעל את החישוב.



איור 8.2: השפעת Stride : למעלה - Stride=1 (דגימה צפופה), למטה - Stride=2 (דגימה גסה יותר).

כעת, בשילוב בין גובה הקלט H, Padding p, גודל הפילטר k וה־Stride s, ממדי הפלט מוגדרים כך:

$$H_{\text{out}} = \left\lfloor \frac{H + 2p - k}{s} + 1 \right\rfloor \tag{8.2}$$

$$W_{\text{out}} = \left\lfloor \frac{W + 2p - k}{s} + 1 \right\rfloor \tag{8.3}$$

Pooling

לאחר קונבולוציה, לעיתים קרובות מבצעים פעולה של Pooling כדי לדגום מחדש את מפת הפיצ'רים. הצורה הנפוצה ביותר היא **Max Pooling** - בחירת הערך הגבוה ביותר מכל טלאי מקומי. לחלופין, **Average Pooling** מחשב את הממוצע באזור.

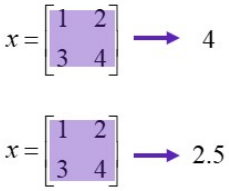
פורמלית, אם $R_{i,j}$ הוא אזור הפולינג עבור מיקום הפלט (i, j), אז:

Max Pooling:

$$P_{\max}(i,j) = \max_{(u,v)\in R_{i,j}} F(u,v) \tag{8.4}$$

Average Pooling:

$$P_{\text{avg}}(i,j) = \frac{1}{|R_{i,j}|} \sum_{(u,v)\in R_{i,j}} F(u,v) \tag{8.5}$$



איור 8.3: גרסאות Pooling. Max Pooling למעלה , למטה Average Pooling. שתיהן מקטינות רזולוציה מרחבית.

בלוק CNN בסיסי

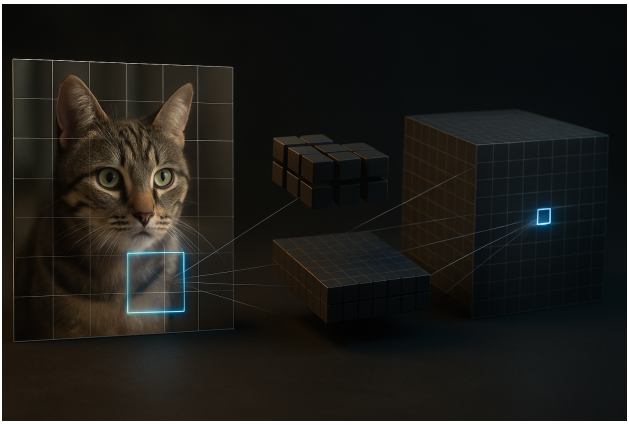
בלוק CNN טיפוסי כולל את השלבים הבאים:

- **קלט:** תמונה, למשל בגודל 28×28 פיקסלים.
- **Convolution :** מימוש פילטרים 3×3 (למשל 8 פילטרים נלמדים).
- **Activation :** יישום פונקציה לא־ליניארית, לרוב ReLU.
- **Pooling :** הקטנת הממדים המרחביים באמצעות Max Pooling.
- **Flatten :** המרת מפת הפיצ'רים לווקטור חד־ממדי.
- **Fully Connected Layer:** ביצוע סיווג סופי על בסיס הפיצ'רים שנחולצו.

בלוקים כאלה מרכיבים יחד מודלים עמוקים יותר. עם כל שכבה נוספת, Receptive Field של הפלט גדל.

Receptive Field ועומק

למרות שכל יחידת קונבולוציה רואה רק אזור קטן, אוסף שכבות רבות מגדילה את ה־Receptive Field - האזור בקלט שמשפיע על נוירון ביציאה.



איור 8.4: ה־ Receptive Field גדל עם עומק - מה שמאפשר למודל ללמוד הקשר גלובלי.

ברשת CNN השכבות התחתונות מזהות קצוות מקומיים, שכבות ביניים מזהות טקסטורות ומוטיבים והשכבות העליונות מגיבות לאובייקטים שלמים.

סיכום

CNN משלבת פילטרים, Stride, Padding ו־Pooling למערכת היררכית חזקה ללמידת פיצ'רים מרחביים. מנגנונים אלה מאפשרים לרשת להקטין את גודל הדאטה, לשמר מבנה ולבנות ייצוגים מופשטים יותר ויותר של קלט חזותי. בשיעור הבא נציג שיפורים ארכיטקטוניים כגון Batch Normalization ו־Dropout המשפרים עוד יותר את הביצועים והרובסטיות.

שיעור 9

רצפים וזמן: RNN , GRU ו־LSTM

מבוא

עד כה עסקנו בקלטים כגון תמונות, שבהם כל הדאטה זמין בבת אחת. אך תחומים רבים - כגון שפה טבעית, מוזיקה וזרמי חיישנים - דורשים עיבוד **דאטה רציף** (Sequential Data). במשימות כאלה ההקשר הטמפורלי חיוני: המודל חייב לא רק "לראות" אלא גם **לזכור**.

בשיעור זה נציג את משפחת רשתות ה־Recurrent Neural Networks (RNNs), נתחיל מה־RNN הבסיסי, ננתח את מגבלותיו ואז נציג שתי ארכיטקטורות נפוצות: GRU ו־LSTM. נסיים עם שיפור עוצמתי - Bi-Directional LSTM.

רשתות RNN

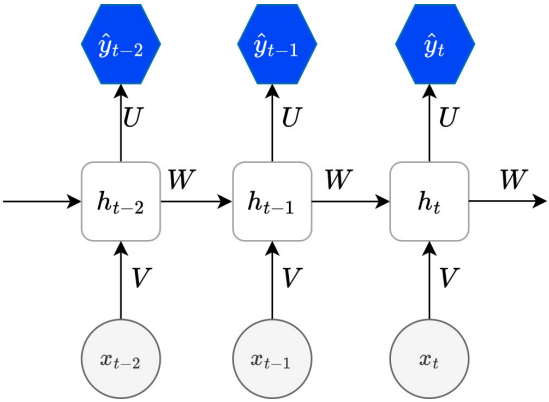
בניגוד לתמונות, שבהן כל הפיקסלים מעובדים בו־זמנית, רצפים נפרשים צעד אחר צעד. בכל צעד זמן t, נכנס קלט חדש x_t והמודל חייב לעדכן את מצבו הפנימי כך שישקף גם את הקלט הנוכחי וגם את ההיסטוריה הקודמת. RNN שומרת מצב חבוי h_t שמתעדכן בצורה רקורסיבית:

$$h_t = f_h(\mathbf{V}x_t + \mathbf{W}h_{t-1} + b_h) \tag{9.1}$$

$$\hat{y}_t = f_y(\mathbf{U}h_t + b_y) \tag{9.2}$$

כאשר:

- x_t הוא הקלט בצעד זמן t.
- h_t הוא המצב החבוי בצעד זמן t.
- $\mathbf{V}, \mathbf{W}, \mathbf{U}$ הן מטריצות משקלים נלמדות.
- b_h, b_y הן הטיות.
- f_h היא פונקציית אקטיבציה (למשל $\tanh$ או ReLU).
- $\hat{y}_t$ הוא הפלט של המודל בצעד t.



איור 9.1: ארכיטקטורת RNN לאורך זמן.

מנגנון זה מאפשר למידע לזרום דרך הזמן. עם זאת, RNNs קלאסיות סובלות מבעיית **Vanishing Gradient**, במיוחד על רצפים ארוכים. הגרדיאנטים דועכים בעת Backpropagation Through Time (BPTT), מה שמגביל את היכולת ללמוד תלות ארוכת טווח.

LSTM: Long Short-Term Memory

רשת Long Short-Term Memory (LSTM) מציגה תא זיכרון פנימי C_t ומערכת **שערים** השולטים במעבר המידע. ארכיטקטורה זו נבנתה כדי להתמודד עם

בעיית ה־Vanishing Gradient. עדכוני LSTM מוגדרים כך:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad \text{(שער קלט)} \tag{9.3}$$

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad \text{(שער שכחה)} \tag{9.4}$$

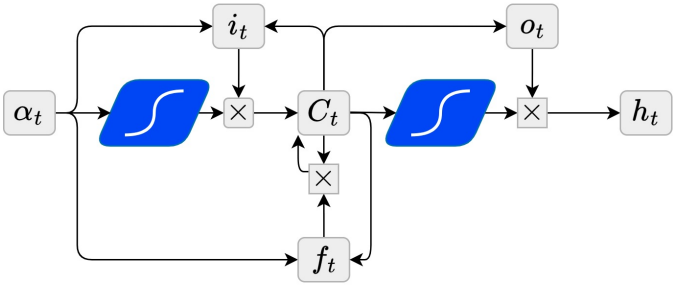
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad \text{(שער פלט)} \tag{9.5}$$

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{9.6}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{9.7}$$

$$h_t = o_t \odot \tanh(C_t) \tag{9.8}$$

כאן σ היא פונקציית סיגמואיד, $\odot$ מסמל כפל איבר־איבר, C_t הוא מצב התא ו־h_t הוא המצב החבוי.



איור 9.2: המבנה הפנימי של יחידת LSTM .

ארכיטקטורה זו מאפשרת זכרון ושכחה סלקטיבית והופכת את ה־LSTM לאידיאלי ללמידת תלות קצרה וארוכה כאחד.

GRU:Gated Recurrent Unit

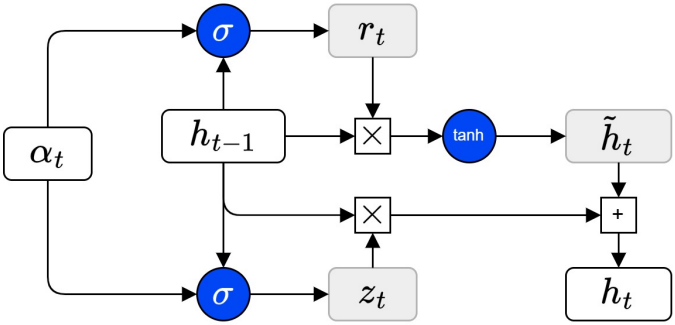
GRU היא גרסה פשוטה יותר של LSTM הממזגת את שער הקלט ושער השכחה לשער עדכון יחיד. יש לה פחות פרמטרים ולעיתים מתאמנת מהר יותר, תוך שמירה על ביצועים דומים. משוואות ה־GRU הן:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \quad \text{(שער עדכון)} \tag{9.9}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \quad \text{(שער איפוס)} \tag{9.10}$$

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{9.11}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{9.12}$$



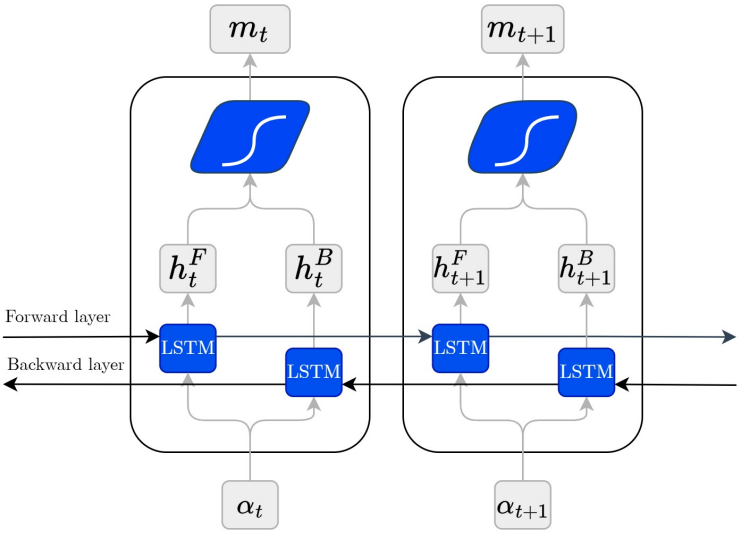
איור 9.3: ארכיטקטורת GRU עם שני שערים: עדכון ואיפוס.

GRUs זולות חישובית ולעיתים מועדפות עבור רצפים ארוכים כאשר זמן האימון מהווה מגבלה.

Bi-Directional LSTM

חלק מהמשימות מרוויחות מגישה גם לעבר וגם לעתיד. Bi-Directional LSTM (Bi-LSTM) משיגה זאת על ידי הרצת שתי LSTMs במקביל: אחת מעבדת קדימה ($t = 1 \to T$) והשנייה אחורה ($t = T \to 1$). הפלטים משולבים בכל צעד זמן:

$$h_t^{\text{bi}} = [h_t^{\rightarrow}; h_t^{\leftarrow}] \tag{9.13}$$



איור 9.4: Bi-LSTM : שילוב מצבים חבויים קדימה ואחורה.

Bi-LSTMs יעילות במיוחד במשימות עיבוד טקסט, שבהן משמעות המילה תלויה לעיתים גם במילים שקדמו וגם באלה שבאו אחריה.

דוגמה: ניתוח סנטימנט

נבחן את המשפט:

"The service was okay, but the food was absolutely terrible."

RNN חד־כיווני המעבד משמאל לימין עשוי לקרוא תחילה את החלק *"The service was okay"* ולהסיק שהסנטימנט נייטרלי או חיובי במעט. אך האות המרכזי מופיע רק בסוף: *"absolutely terrible"*, המעיד על שליליות חזקה.

RNN חד־כיווני עלול להיכשל בתיקון התחזית המוקדמת. לעומת זאת, **Bi-LSTM** מעבדת את הרצף בשני הכיוונים. היא יכולה לשלב את ההקשר

מהסוף ("terrible") בהבנתה את המשפט כולו - וכך להגיע לסיווג סנטימנט מדויק יותר. דוגמה זו מדגימה את התלות בהקשר.

מודל	שערים	זיכרון	יתרונות	שימוש
RNN	אין	קצר טווח	פשוט	רצפים קצרים
LSTM	קלט, שכחה, פלט	ארוך טווח	יציב, חזק	טקסט, אודיו
GRU	עדכון, איפוס	ארוך טווח	מהיר, פחות פרמטרים	כמו LSTM

טבלה 9.1: סיכום ארכיטקטורות למידע רציף.

סיכום

מודלים רציפים חיוניים למשימות הכוללות דאטה בזמן. ה־RNN הבסיסי מציג את עקרון הזיכרון לאורך זמן. LSTM ו־GRU משפרים זאת על ידי מתן יכולת לרשת לשמור ולשלוט במידע בצורה יעילה יותר.

במשימות הדורשות הקשר עתידי, Bi-LSTM מרחיבה יכולת זו. עם זאת, ארכיטקטורות אלו עדיין נשענות על חישוב סדרתי, מה שמגביל את הסקלאביליות שלהן.

בשיעורים הבאים נציג את ארכיטקטורת ה־ **Transformer** ששולטת כיום בשיטות המתקדמות ביותר לעיבוד שפה ומשימות רצף.

שיעור 10

הורדת מימדים עם Autoencoders

מבוא

עד כה התמקדנו בלמידה מונחית (Supervised Learning) - אימון מודלים על זוגות קלט־פלט עם תוויות. בשיעור זה נחקור פרדיגמה אחרת: **למידה לא מונחית** (Unsupervised Learning), שבה אין תוויות נתונות והמטרה היא לחשוף מבנה חבוי בתוך הדאטה.

באופן כללי, למידת מכונה מחולקת לשלושה סוגים מרכזיים:

(1) *Supervised Learning* - המודל ממפה קלטים x לפלטים ידועים y;

(2) *Unsupervised Learning* - מציאת דפוסים או דחיסה ללא תוויות;

(3) *Reinforcement Learning* - סוכן הלומד דרך אינטראקציה ותגמול.

אמנם הקורס מתמקד בלמידה מונחית, אך מודלים לא־מונחים חיוניים באותה מידה. ביניהם, **Autoencoder (AE)** ממלא תפקיד מרכזי בלמידה עמוקה מודרנית. Autoencoder הוא רשת נוירונים המתאמנת לשחזר את הקלט שלה - ובתהליך זה היא לומדת ייצוג פנימי של הדאטה.

ארכיטקטורת Autoencoder

Autoencoder מורכב משתי תת־רשתות:

- **Encoder** f_{enc}: דוחס את הקלט $x \in \mathbb{R}^d$ לווקטור לטנטי $z \in \mathbb{R}^k$, כאשר $k \ll d$.

- **Decoder** f_{dec}: משחזר את הקלט $\hat{x} \in \mathbb{R}^d$ מתוך z.

פורמלית, הפונקציות מקיימות:

$$z = f_{\text{enc}}(x) \tag{10.1}$$

$$\hat{x} = f_{\text{dec}}(z) \tag{10.2}$$

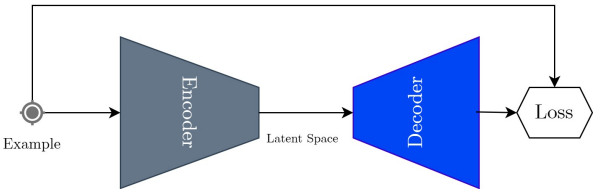
הרשת מתאמנת למזער את שגיאת השחזור:

$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2 = \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 \tag{10.3}$$

פונקציית מחיר זו מעודדת את המודל לשמר מידע חיוני ב־z, תוך השלכת מידע לא רלוונטי.

למידת ייצוגים קומפקטיים

על ידי שימוש במבנה של צוואר בקבוק (כלומר $k \ll d$), ה־Autoencoder נאלץ ללמוד רק את הדפוסים המשמעותיים ביותר בקלט. ה־Encoder מחלץ פיצ'רים מופשטים וה־Decoder לומד להרכיב אותם מחדש לקירוב מדויק של הקלט המקורי. הדבר יעיל במיוחד עבור דאטה עתיר ממדים כמו תמונות, אודיו ואותות חיישנים, שבהם דחיסה ויכולת הפירוש חשובות.



איור 10.1: המחשת Autoencoder .

דוגמה: הורדת ממדים ב- MNIST

להדגמת הרעיון, ניישם Autoencoder על דאטהסט MNIST - המורכב מתמונות גווני אפור בגודל $28 \times 28 = 784$ פיקסלים של ספרות בכתב יד. נבנה Autoencoder עם:

- קלט: $x \in \mathbb{R}^{784}$
- מרחב לטנטי: $z \in \mathbb{R}^2$
- פלט: $\hat{x} \in \mathbb{R}^{784}$

מספר הפרמטרים הנלמדים ברשת זו הוא 218,514.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 64)	8,256
dense_2 (Dense)	(None, 2)	130
dense_3 (Dense)	(None, 64)	192
dense_4 (Dense)	(None, 128)	8,320
dense_5 (Dense)	(None, 784)	101,136

איור 10.2: ארכיטקטורת Autoencoder עבור MNIST.

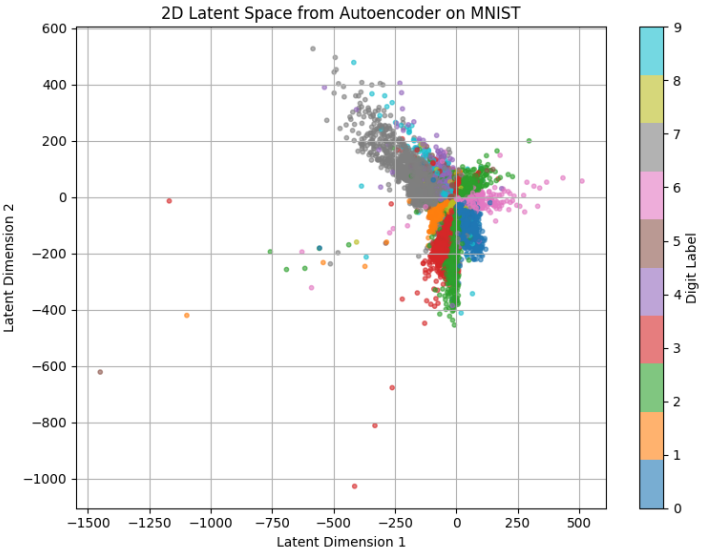
לאחר האימון, אנו מחלצים את הווקטורים הלטנטיים z עבור כל הספרות. למרות שלא ניתנו תוויות במהלך האימון, ספרות עם מבנה דומה (למשל "1" ו־"7") מקובצות יחד.

דבר זה מדגים שהמודל הצליח ללמוד את המבנה הפנימי של הדאטה - סימן מובהק ללמידת ייצוג מוצלחת.

תוצאות השחזור

נציג שחזורי ספרות מתוך ה־Autoencoder. לכל תמונת קלט x, נעביר אותה ב- Encoder-Decoder ונשווה לשחזור $\hat{x}$:

על אף צוואר הבקבוק, המודל מצליח לשמר את רוב המבנה, מה שמראה כי חילץ פיצ'רים לטנטיים משמעותיים.



איור 10.3: מרחב לטנטי דו־ממדי שנלמד על ידי Autoencoder. הצבעים מציינים תוויות אמיתיות (לצורכי ויזואליזציה בלבד).

עקומת האימון

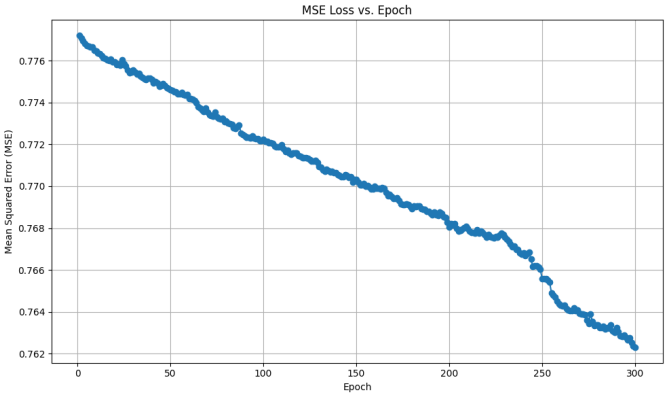
האיור הבא מציג את שגיאת השחזור (MSE) כפונקציה של מספר האפוקים. ככל שהאימון נמשך, השגיאה יורדת בהדרגה:

השוואה ל־ PCA

Principal Component Analysis (PCA) היא טכניקה ליניארית לצמצום ממדים. היא מוצאת כיוונים אורתוגונליים הממקסמים וריאנס. עם זאת, PCA אינה יכולה ללמוד אי־ליניאריות בדאטה.

Autoencoders מכלילים את PCA באמצעות פונקציות לא־ליניאריות (למשל אקטיבציית ReLU), מה שמאפשר להם ללמוד יריעות מעוקלות להפרדת המחלקות למשל. ההבדל המרכזי הוא ש־PCA מוצאת תת־מרחב ליניארי מינימלי בשגיאת שחזור ריבועית, בעוד ש־Autoencoders לומדים פונקציה לא־ליניארית הממזערת אותה פונקציה. Autoencoders גמישים יותר ומתאימים

איור 10.4: ספרות מקוריות (למעלה) ושחזור (למטה) לאחר אימון Autoencoder ל־300 אפוקים.



איור 10.5: שגיאת שחזור MSE לעומת אפוקים. המודל מתכנס ביציבות לשגיאה נמוכה.

במיוחד לדאטה מובנה כמו תמונות.

יישומים והרחבות

Autoencoders שימשו בהצלחה במגוון רחב של משימות למידה עמוקה. כמה מהיישומים המרכזיים כוללים:

- למידת ייצוגים בממד נמוך מקלטים במדד גבוה.
- דחיסת אותות לאחסון או שידור יעיל.
- זיהוי חריגות/אנומליות בהתבסס על שגיאת שחזור גבוהה.
- פרה־טריינינג לרשתות עמוקות ולאחריו Fine-Tuning מונחה.

- ניקוי רעשים מהקלט (Denoising).

מעבר ל־Autoencoder הבסיסי, פותחו הרחבות רבות לצרכים ספציפיים. אחת מהן היא Variational Autoencoder (VAE), שבה ה־Encoder מפיק פרמטרים של התפלגות גאוסית - וקטור ממוצעים $\mu(x)$ ווקטור סטיות תקן $\sigma(x)$ - שממנו נדגם המשתנה הלטנטי z כך:

$$z \sim \mathcal{N}(\mu(x), \sigma^2(x)). \tag{10.4}$$

סיכום

Autoencoders הם כלי יסודי בלמידה לא־מונחית, החושף את המבנה הפנימי של דאטה באמצעות דחיסה ושחזור. הם אינם נשענים על תוויות ובכל זאת לומדים ייצוגים משמעותיים שניתן להמחיש, לקבץ ולהשתמש בהם במשימות המשך.

הדוגמה על MNIST ממחישה כיצד Autoencoders חושפים תבניות גיאומטריות במרחב הספרות מבלי שנאמר להם מהי ספרה. כוחם טמון ביכולת להכליל וללמוד את מהות המבנה של הדאטה. בכך הם הופכים לכלי בלתי נפרד במערכות למידה עמוקה מודרניות.

שיעור 11

Self-Attention וה- Transformer

מבוא

ארכיטקטורת ה־Transformer, שהוצגה בשנת 2017 במאמר Attention Is All You Need, חוללה מהפכה בלמידה עמוקה עבור דאטה רציף. במקור פותחה עבור Natural Language Processing (NLP), אך במהרה הפכה ליסוד של מודלים ג'נרטיביים מודרניים כגון GPT ו־BERT. החידוש המרכזי שלה - Self-Attention - החליף את הרקורסיה במנגנון המאפשר לכל טוקן להתייחס לכל שאר הטוקנים, מה שאפשר מיקבול רחב יותר ומידול טוב יותר של תלות לטווח ארוך. מאוחר יותר הורחב עקרון זה גם לתחומים אחרים: למשל, Vision Transformers (ViT) על תמונות.

מדוע RNNs לא מספיקות

רשתות RNN וגרסאותיהן כמו LSTM מעבדות רצפים טוקן אחרי טוקן, תוך עדכון המצב החבוי השומר מידע מהעבר. גישה סדרתית זו מקשה על מיקבול ומגבילה את היכולת ללמוד תלות רחוקה.

> **דוגמה:** *"The book that the professor recommended at the end of the lecture was truly brilliant."* כדי לפרש נכון את המילה "brilliant" כמתייחסת ל־"book", על המודל ללמוד תלות ארוכת טווח - משהו ש־RNN מתקשה לשמר.

מנגנון ה־Attention

Self-Attention פותר זאת על ידי כך שכל טוקן יכול להתייחס ישירות לכל הטוקנים האחרים ברצף. נניח $X = [x_1, x_2, \ldots, x_T]$, כאשר כל $x_i \in \mathbb{R}^d$ הוא ה־Embedding של הטוקן ה־i.

כל וקטור קלט מועבר באופן ליניארי לשלושה ייצוגים:

$$q_i = W_Q x_i \tag{11.1}$$

$$k_i = W_K x_i \tag{11.2}$$

$$v_i = W_V x_i \tag{11.3}$$

כאשר:

- $q_i \in \mathbb{R}^{d_k}$ הוא וקטור ה־**Query** של הטוקן i.
- $k_i \in \mathbb{R}^{d_k}$ הוא וקטור ה־**Key** של הטוקן i.
- $v_i \in \mathbb{R}^{d_v}$ הוא וקטור ה־**Value** של הטוקן i.
- $W_Q, W_K, W_V \in \mathbb{R}^{d_k \times d}$ הן מטריצות נלמדות.

ציון ה־Attention בין טוקן i לטוקן j מחושב באמצעות מכפלה מנורמלת:

$$\alpha_{ij} = \frac{q_i^\top k_j}{\sqrt{d_k}} \tag{11.4}$$

לאחר מכן מחושבים משקלי ה־Attention באמצעות פונקציית Softmax:

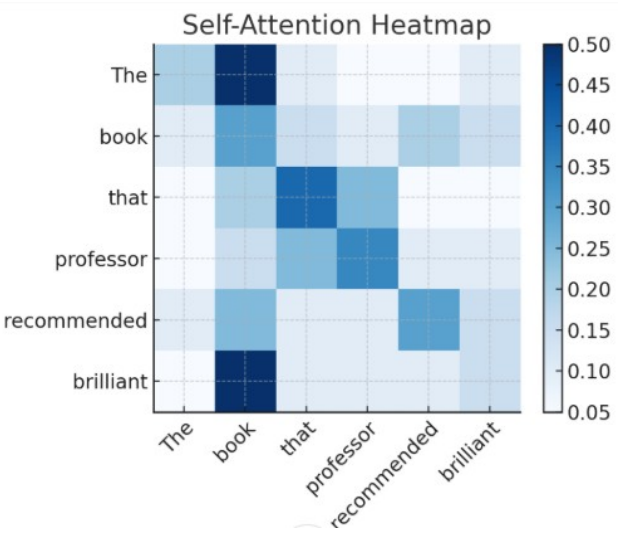
$$a_{ij} = \frac{\exp(\alpha_{ij})}{\sum_{j'=1}^{T} \exp(\alpha_{ij'})} \tag{11.5}$$

לבסוף, כל ערך מתעדכן על ידי סכום משוקלל של וקטורי ה־Value:

$$z_i = \sum_{j=1}^{T} a_{ij} v_j \tag{11.6}$$

תהליך זה יוצר ייצוג מותאם־הקשר z_i עבור כל טוקן.

נביט לדוגמה במפת Self-Attention עבור המשפט: *"The solution that the professor recommended was brilliant."* כל שורה מייצגת את ה־Query של מילה והעמודות מייצגות את המילים שאליהן היא מתייחסת. תאים כהים יותר מצביעים על משקל Attention גבוה יותר. לדוגמה, המילה "book" מתייחסת בחוזקה ל־"brilliant" וכך נשמרת ההתאמה למרות טוקנים שישנם רבים המופיעים ביניהם.

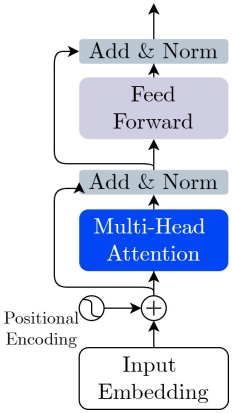


איור 11.1: מפת Self-Attention עבור המשפט: *"The solution that the professor recommended was brilliant."*

מה־Attention ל־Transformer

Transformer מלא מורכב מ־**ארכיטקטורת Encoder-Decoder**. ה־**Encoder** ממפה רצף קלט (למשל משפט) לרצף ייצוגים חבויים באמצעות שכבות של Self-Attention ורשתות Feedforward. ה־**Decoder** משתמש ב־Masked Self-Attention (כדי למנוע הצצה קדימה), ב־Cross-Attention עם פלט ה־Encoder וברשת Feedforward כדי ליצור את פלטי הטוקנים אחד־אחד.

מבנה זה מאפשר הפקה מותנית, מה שהופך את ה־Transformer למודל ג'נרטיבי עוצמתי - מתאים למשימות כגון תרגום, יצירת טקסט, כתיבת תיאורי תמונות ועוד.



איור 11.2: המחשת Encoder

האיור מציג את המבנה הפנימי של בלוק Transformer Encoder. בליבו נמצא מודול Multi-Head Attention - הרחבה מקבילית של מנגנון ה־Attention - המאפשר למודל להתייחס לתת־מרחבים שונים של ייצוג הקלט בו־זמנית.

פלט ה־Attention עובר דרך Residual Connection, נרחיב עליו בהמשך, ואחריו Layer Normalization (המכונה "Add & Norm"), המייצבת את התפלגות האקטיבציות ומאפשרת ארכיטקטורות עמוקות יותר. לאחר שכבת ה־Attention באה רשת Position-Wise Feedforward, בדרך כלל שתי שכבות מלאות עם פונקציית אקטיבציה לא־ליניארית (כגון ReLU או GELU).חיבור שאריות נוסף ושלב נורמליזציה משלימים את הבלוק.

חשוב לציין שארכיטקטורה זו מאפשרת מידול הקשר תלוי־סדר מלא ללא שימוש ברקורסיה או קונבולוציה, תוך הסתמכות בלעדית על דפוסי Attention ונלמדים ועל Positional Encodings. מבנה זה מהווה את עמוד השדרה של מודלים שפתיים מודרניים.

יתרונות והכללות

ל־Transformers יתרונות מרכזיים על פני ארכיטקטורות קודמות: הם מאפשרים מיקבול מלא של כל רכיבי הרצף - מה שמאיץ אימון. מנגנוני ה־Self-Attention ממדלים באופן טבעי הקשר גלובלי ותלות ארוכת טווח, בניגוד ל־RNNs המתקשים בכך. הם ניתנים להעמקה רבה (Deep Networks) ותומכים בהעברת ידע דרך מודלים מאומנים מראש (Pretrained Language Models).

סיכום

Self-Attention שינתה מהיסוד את ארכיטקטורת המודלים בלמידה עמוקה. באמצעות מתן אפשרות לכל טוקן לאסוף מידע מכל הרצף, נוצרו מערכות חזקות וסקלאביליות שתרמו לתחומים רבים. ארכיטקטורת ה־Transformer, עם מבנה ה־Encoder-Decoder, מהווה את הבסיס למודלים הג'נרטיביים המובילים כיום בתחום עיבוד השפה הטבעית ואף מעבר.

שיעור 12

Normalization ו־ Initialization

מבוא

בשיעור זה נחקור שתי שיטות קריטיות בלמידה עמוקה: **Normalization** ו־**Initialization** . טכניקות אלו חיוניות לייצוב והאצת האימון של רשתות נוירונים עמוקות. ללא נורמליזציה ואתחול נכונים, מודלים עלולים להתכנס לאט, להיתקע בנקודות מינימום מקומיות או לסבול מבעיות של Vanishing Gradients או Exploding Gradients. נתחיל בהבנת הצורך לנרמל אקטיבציות ברשתות עמוקות ולאחר מכן נדון כיצד בחירת משקולות התחלתיות משפיעה על מעבר המידע ברשת.

מדוע לנרמל?

במהלך האימון, האקטיבציות זורמות דרך הרשת. עם זאת, ככל שהעומק גדל, ההתפלגות שלהן עשויה להשתנות בצורה דרמטית בין שכבות או מיני־באצ'ים. תופעה זו, הידועה בשם Internal Covariate Shift, מפריעה לתהליך הלמידה וגורמת לתנודות בגרדיאנטים. כדי למתן זאת, משתמשים בטכניקות נורמליזציה כגון **Batch Normalization** ו־ **Layer Normalization**.

Batch Normalization

בהינתן מיני־באץ' של אקטיבציות $\{x_1, x_2, \ldots, x_m\}$ מנוירון מסוים (או ערוץ פיצ'ר), Batch Normalization מנרמל את הערכים על פני ממד הבאץ':

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{12.1}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{12.2}$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{12.3}$$

$$y_i = \gamma \hat{x}_i + \beta \tag{12.4}$$

כאשר:

- μ_B ו־σ_B^2 הם הממוצע והשונות המחושבים על פני המיני־באץ'. $\hat{x}_i$ היא האקטיבציה המנורמלת.
- γ ו־β הם פרמטרים נלמדים המאפשרים שינוי קנה מידה והזזה. ε הוא קבוע קטן להבטחת יציבות נומרית.

בדרך כלל מוסיפים BatchNorm אחרי הטרנספורמציה הליניארית ולפני פונקציית האקטיבציה:

$$\text{Linear} \rightarrow \text{BatchNorm} \rightarrow \text{ReLU}$$

שיטה זו משפרת את חישוב הגרדיאנטים, מאפשרת אימון מהיר יותר ואף מאפשרת שימוש בקצב למידה גבוה יותר. עם זאת, היא מסתמכת על מיני־באצ'ים מספיק גדולים לסטטיסטיקה יציבה.

Layer Normalization

במשימות עם מיני־באצ'ים קטנים מאוד או ארכיטקטורות מבוססות רצף (כמו Transformers), BatchNorm עלול להיות פחות יעיל או מודלים לשפה טבעית), בשל סטטיסטיקות באצ' לא יציבות. במקרים אלו, **Layer Normalization** מתאים יותר, משום שהוא מנרמל על פני הפיצ'רים של כל דוגמה בודדת - ולא על פני הבאצ'.

בהינתן וקטור קלט $x \in \mathbb{R}^d$ (למשל האקטיבציות של שכבה אחת עבור דוגמה בודדת), החישוב הוא:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i \tag{12.5}$$

$$\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2 \tag{12.6}$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}} \tag{12.7}$$

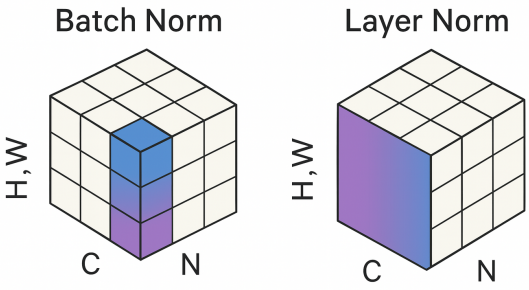
$$y_i = \gamma \hat{x}_i + \beta \tag{12.8}$$

כאן, הנורמליזציה מבוצעת על פני כל d הפיצ'רים של הדוגמה היחידה ואותם פרמטרי קנה מידה והזזה נלמדים (γ, β).

שיטה זו אינה תלויה בגודל הבאצ' ועובדת היטב במודלים שבהם נדרשת עקביות לכל דוגמה, כמו ברשתות רצף (RNNs) או דקודרים אוטורגרסיביים. היא אפקטיבית במיוחד ב־Transformers וברשתות רצף שבהן תבניות הקלט משתנות ותלויות בזמן.

Weight Initialization

לפני תחילת האימון, המשקולות מאותחלות לרוב בערכים אקראיים. עם זאת, אתחול לא נכון עלול לעוות את זרימת המידע, לגרום ל־Vanishing/Exploding Gradients ולעכב את הלמידה. המטרה היא לשמר שונות מבוקרת של



איור 12.1: השוואה בין Batch Norm (משמאל) ו־Layer Norm (מימין). Batch Norm מנרמל על פני ממד הבאץ' (N), בעוד Layer Norm מנרמל על פני כל הפיצ'רים בדוגמה אחת (C, H, W).

האקטיבציות והגרדיאנטים בכל השכבות.

Xavier Initialization

שיטה זו מתאימה לאקטיבציות סימטריות כמו tanh או sigmoid. **Xavier Initialization** דוגמת משקולות מהתפלגות עם שונות:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right) \tag{12.9}$$

כאשר n_{in} ו־n_{out} הם מספר הניורונים הנכנסים והיוצאים בשכבה. שיטה זו מאזנת את מעבר המידע קדימה ואחורה ברשת כדי למנוע דעיכה או הגדלה יתר על המידה של הנתונים.

He Initialization

עבור אקטיבציות לא־סימטריות כמו ReLU, שמאפסות קלטים שליליים, **He Initialization** מגדילה את השונות ל־:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right) \tag{12.10}$$

כך נשמר מעבר הגרדיאנטים למרות הדילול שנגרם על ידי ReLU.

סיכום

רשתות עמוקות רגישות לסקאלה ולהתפלגות של אקטיבציות וגרדיאנטים. נורמליזציה - ובמיוחד BatchNorm ו־LayerNorm - מטפלת בחוסר יציבות על ידי סטנדרטיזציה של ערכי הפיצ'רים ובכך משפרת את מעבר המידע בין שכבות. במקביל, אתחול נכון מבטיח שהגרדיאנטים לא יהיו גדולים מדי ולא קטנים מדי בתחילת האימון.

אתחול Xavier אידיאלי לרשתות עם אקטיבציות סימטריות, בעוד אתחול He מותאם למודלים מבוססי ReLU. סכמות אלו ממזערות עיוות נתונים ומשפרות את יעילות הלמידה.

ביחד, נורמליזציה ואתחול מהווים את עמוד השדרה של דינמיקת אימון יציבה, מהירה ורובסטית בלמידה עמוקה מודרנית. בשיעור הבא נחקור כיצד להעשיר דאטה עבור אימון בצורה יעילה (Data Augmentation), כדי לשפר הכללה מבלי לאסוף דוגמאות נוספות.

שיעור 13

אוגמנטצייה

מבוא

מודל עשוי להתאמן ביציבות, להתכנס בצורה חלקה ולהציג Training Loss נמוך - ובכל זאת להיכשל בצורה דרמטית במימוש. מדוע זה קורה?

בעולם האמיתי, דאטה הוא רועש, לא עקבי ולעיתים רחוקות דומה לדוגמאות המטופחות ששימשו אותנו באימון. המפתח להכללה - היכולת של המודל להצליח על קלטים שלא ראה קודם - אינו תמיד חבוי בארכיטקטורה עצמה. לעיתים קרובות הוא טמון בדאטה. **Data Augmentation** הוא התהליך של הרחבה והעשרת הדאטהסט על ידי מימוש טרנספורמציות השומרות על נכונות התוויות. הוא מגדיל רובסטיות על ידי סימולציה של שונות מהעולם האמיתי וכופה על המודל להפוך לאינווריאנטי לשינויים לא רלוונטיים.

מתי Augmentation עוזר?

Data Augmentation שימושי במיוחד כאשר דאטהסט האימון קטן, לא מאוזן, או חסר גיוון. במצבים כאלה המודל נוטה ל־Overfitting - השגת דיוק גבוה על דאטהסט האימון אך כישלון בהכללה לדוגמאות חדשות. אוגמנטציה מכניסה וריאציות מבוקרות לדאטה ועוזרת למודל להפוך לרובסטי לשינויים בהתפלגות. זה חשוב במיוחד כאשר הדומיין של הבדיקה כולל דפוסים או עיוותים שאינם מיוצגים היטב בדאטהסט האימון. אסטרטגיית Augmentation

טובה תגרום בדרך כלל לעלייה קלה ב־Training Loss, כי המודל מתמודד עם קלטים קשים יותר ומגוונים יותר, אך תוביל לירידה ניכרת ב־Validation Loss ולשיפור ב־Accuracy על דאטה שהמודל לא ראה. כתוצאה מכך, Augmentation מהווה שיטה פשוטה אוחזקה לשיפור הכללה ואמינות, במיוחד במשימות ראייה או כאשר הדאטה המתויג מוגבל.

Augmentation לעומת Generation

חשוב להבחין בין Augmentation ל־Generation. Augmentation אינו מייצר דוגמאות חדשות לחלוטין. במקום זאת, הוא יוצר גרסאות חלופיות של אותה דוגמה תוך שמירה על התווית הסמנטית.

נניח כי x היא תמונת קלט ו־y התווית שלה. Augmentation מייצר גרסה מומרת $x' = T(x)$ כך ש־:

$$f(x') \approx f(x), \quad \text{and} \quad \text{label}(x') = y \tag{13.1}$$

כאשר T היא פונקציית טרנספורמציה שנלקחת מהתפלגות של "שיבושים" לגיטימיים ו־f הוא המודל.

שמירת עקביות התווית

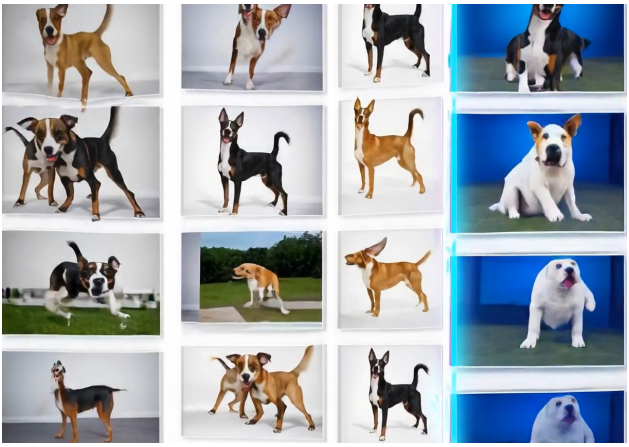
דרישה בסיסית היא שהדוגמה המומרת חייבת לשמור על התווית המקורית. הפרת כלל זה עלולה לגרום תיוג רועש" ולביצועים גרועים.

למשל, סיבוב הספרה "6" ב־$180°$ יוצר משהו שקשה להבחין בינו לבין "9" - אך סימונו כ־"6" יטעה את המודל. זה כבר לא Augmentation, אלא הרעשה והטעיה סמנטית.

טרנספורמציות נפוצות

בתחום הוויזואלי, נהוגות הטרנספורמציות הבאות:

- **Horizontal Flip**: $x' = \text{Flip}_H(x)$ - מתאים כאשר הסימטריה מאפשרת.
- **Rotation**: $x' = \text{Rotate}_\theta(x),\ \theta \in [-15°, 15°]$
- **Crop and Resize**: חיתוך אקראי של אזור בתמונה והחזרה לגודל המקורי.
- **Translation and Zoom**: הזזות קטנות ושינוי קנה מידה מבלי לשנות זהות מחלקה.
- **Brightness and Contrast**: התאמות ברמות תאורה וניגודיות.
- **Color Jitter**: שינויי גוון/רוויה לתמונות צבע.
- **Additive Noise**: הוספת רעש גאוסי.



איור 13.1: דוגמאות ל־Augmentation .

זהירות

עודף Augmentation עלול להזיק. אם הדוגמאות המתקבלות כבר אינן דומות לקלטים תקפים - למשל עיוות קיצוני או אובדן מבנה אובייקט - הדבר עלול לפגוע בלמידה. בדיקה ויזואלית חיונית.

מימוש

ספרייה מומלצת ל־Image Augmentation היא **Albumentations**, המשתלבת היטב הן עם PyTorch והן עם TensorFlow. ניתן לבצע באופן אקראי היפוכים, סיבובים, שינויי בהירות ונורמליזציה - תוך שמירה על הסמנטיקה של התווית.

Offline לעומת Online Augmentation

ישנן שתי אסטרטגיות ליישום Augmentation:

- **Offline Augmentation**: יוצר ושומר דוגמאות חדשות מראש. מגדיל שימוש בזיכרון אך מספק וריאציה קבועה.
- **Online Augmentation**: מיישם טרנספורמציות אקראיות בזמן אמת במהלך האימון. מספק סטוכסטיות וגיוון אינסופי לאורך אפוקים.

מרבית המערכות המודרניות מעדיפות Online Augmentation בשל הגמישות ויעילות הזיכרון.

Best Practices

כדי למקסם את האפקטיביות של Augmentation:

1. **Validate visually**: תמיד בדקו באצ' של תמונות מומרות.
2. **Prioritize plausibility**: הימנעו מטרנספורמציות שמפרות זהות מחלקה.
3. **Prefer small, composable transformations**: עדיף לשרשר פעולות פשוטות מאשר לבצע עיוות אגרסיבי יחיד.
4. **Adapt per domain**: Augmentation לספרות אינו זהה לזה לתמונות רפואיות או לתמרורי תנועה.

סיכום

Augmentation היא אחת הטכניקות החזקות והנגישות ביותר לשיפור רובסטיות של מודלים. היא מדמה שונות, כופה אינווריאנטיות ומפחיתה תלות ברגולריזציה ידנית. בניגוד לשינויים בארכיטקטורה או לפונקציות מחיר חדשות, אוגמנטציה משנה את הדאטה עצמו - ועוזרת למודל לפגוש את העולם כפי שהוא באמת: רועש, מגוון ובלתי צפוי.

אוגמנטציה מתוכננת היטב אינה רק מעשירה את דאטהסט האימון - אלא מרחיבה את נקודת המבט של המודל. היא מכינה את הרשת לא לשנן, אלא להבין.

שיעור 14

אופטימיזציה מתקדמת

למידה עמוקה מודרנית לא הייתה אפשרית ללא אלגוריתמי אופטימיזציה יעילים ויציבים. בעוד שרשתות נוירונים עשויות להיראות טובות בתחזיותיהן, בליבתן הן נשענות על עדכונים חוזרים וזהירים למיליוני פרמטרים. בשיעור זה נבחן כיצד מתבצעים עדכונים אלו - מהעיקרון הבסיסי של Gradient Descent ועד האופטימייזרים המתקדמים ביותר כיום: AdamW, Lion ו־Adafactor. אופטימייזרים אלו קובעים כיצד המודל לומד, באיזו מהירות הוא מתכנס והאם הוא מצליח להכליל.

מ־Backpropagation לאופטימיזציה

כל צעד אימון מתחיל ב־Backpropagation, שמחשב את הגרדיאנטים של פונקציית המחיר ביחס לכל משקל ברשת. גרדיאנטים אלו מצביעים על הכיוון שבו המחיר גדל ביותר. כדי לשפר את המודל, אנו רוצים ללכת בכיוון ההפוך (מזעור המחיר). האופטימייזר הוא האלגוריתם שהופך גרדיאנטים לעדכוני פרמטרים משמעותיים.

Gradient Descent: חזרה לבסיס

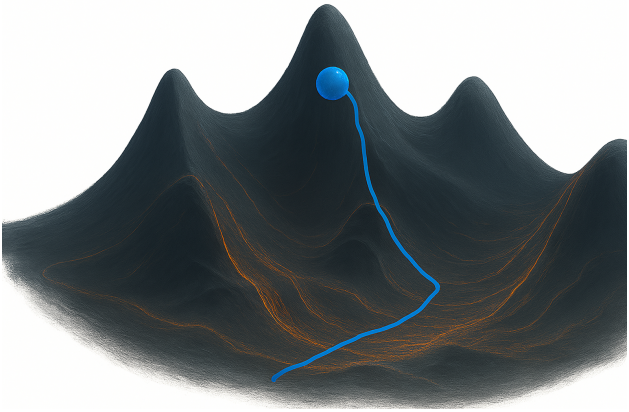
כפי שראינו קודם, כלל העדכון הקלאסי פשוט ויסודי:

$$\theta_{k+1} = \theta_k - \eta \cdot \nabla_\theta L(\theta_k) \tag{14.1}$$

כאשר θ_k מייצג את וקטור הפרמטרים בצעד k, η הוא קצב הלמידה ו־$\nabla_\theta L(\theta_k)$ הוא הגרדיאנט של פונקציית המחיר L בנקודה זו. במילים פשוטות - המודל "גולש" על פני משטח המחיר, מונחה על ידי השיפוע המקומי.

הערה: זו אותה משוואה שראינו קודם, אך כאן המשקל מסומן ב- θ במקום w.

בפועל, משטחי אימון אינם חלקים; הם רועשים, עתירי ממדים ומלאים ב"עמקים" ו"ערוצים". Gradient Descent פשוט מתקשה להתמודד עם מורכבות זו - ולכן יש צורך באופטימייזרים מתקדמים יותר.



איור 14.1: אילוסטרציה של Gradient Descent: כדור מתגלגל על פני משטח מחיר מורכב, מונחה על ידי השיפוע התלול ביותר בכל נקודה. המסלול מייצג את תהליך האופטימיזציה כאשר המודל מתאים את פרמטריו כדי למזער את השגיאה. פסגות ועמקים מדגימים את האתגר ביציאה מנקודות מינימום מקומיות לעבר מינימום גלובלי.

Adam: Adaptive Moments

Adam - Adaptive Moment Estimation - משפר את Gradient Descent הרגיל על ידי שילוב שני מנגנונים: Momentum וסקיילינג אדפטיבי. הוא שומר ממוצע

מעריכי דועך של הגרדיאנטים (מומנט ראשון) ושל הריבוע שלהם (מומנט שני):

$$m_k = \beta_1 \cdot m_{k-1} + (1 - \beta_1) \cdot g_k \tag{14.2}$$

$$v_k = \beta_2 \cdot v_{k-1} + (1 - \beta_2) \cdot g_k^2 \tag{14.3}$$

$$\theta_{k+1} = \theta_k - \eta \cdot \frac{m_k}{\sqrt{v_k} + \epsilon} \tag{14.4}$$

כאשר g_k הוא הגרדיאנט בצעד k ו־m_k, v_k הם ממוצעים נעים של הגרדיאנט ושל ריבועו. האיבר ϵ (בדרך כלל 10^{-8}) מבטיח יציבות נומרית. ברירות מחדל מקובלות: $\beta_1 = 0.9$, $\beta_2 = 0.999$.

Adam מתאים את גודל הצעד לכל פרמטר בנפרד, מה שהופך אותו לרובסטי ויעיל במגוון רחב של משימות. הוא עדיין נחשב לאופטימייזר ברירת המחדל אצל רבים.

AdamW: Decoupled Weight Decay

AdamW הוצג כדי לתקן בעיה עדינה במימוש הרגולריזציה של Adam. ב־Adam, Weight Decay (L2 Regularization) מתקשר בצורה לא רצויה עם קצבי הצעד האדפטיביים. AdamW מפריד ביניהם ומיישם Weight Decay בצורה מפורשת:

$$\theta_{k+1} = \theta_k - \eta \cdot \left(\frac{m_k}{\sqrt{v_k} + \epsilon} + \lambda \cdot \theta_k \right) \tag{14.5}$$

כאן, λ הוא מקדם Weight Decay (למשל 0.01). ניסוח זה משפר הכללה ויציבות באימון, במיוחד במודלים גדולים מבוססי Transformers. כתוצאה מכך, AdamW הוא האופטימייזר המועדף בארכיטקטורות מתקדמות רבות.

Lion: Lightweight Momentum with Signs

Lion, שהוצג ב־2023 על ידי חוקרים בגוגל, מציע טוויסט אלגנטי: במקום

להשתמש בעוצמת הגרדיאנטים או במומנטים שניים, הוא מעדכן משקולות על בסיס **סימן** המומנטום בלבד:

$$m_k = \beta_1 \cdot m_{k-1} + (1 - \beta_1) \cdot g_k \tag{14.6}$$

$$\theta_{k+1} = \theta_k - \eta \cdot \text{sign}(m_k) \tag{14.7}$$

גשיה זו מקטינה שימוש בזיכרון ועלות חישובית, תוך שמירה על הכיוון. Lion יעיל במיוחד במודלים ויזואליים כמו ViTs והוכח כעוקף את AdamW באימון מסוימים, במיוחד כאשר זיכרון ה־GPU מוגבל. היפר־פרמטרים מקובלים: $\beta_1 = 0.9, \quad \beta_2 = 0.99 \quad \eta = 3 \times 10^{-4}$.

Adafactor: יעילות זיכרון בקנה מידה גדול

Adafactor פותח עבור מודלים עצומים, שבהם אחסון מומנטים שניים מלאים הופך לצוואר בקבוק. במקום לשמור מטריצות מלאות, Adafactor מפרק אותן לווקטורים של שורות ועמודות - מצמצם את דרישת הזיכרון מ־$O(n^2)$ ל־$O(n)$. כך ניתן לאמן מודלי שפה ענקיים (כמו T5) גם על חומרה מוגבלת. Adafactor תומך גם במנגנון קצב למידה אדפטיבי לפי סקייל הפרמטרים וניתן לשלבו עם שיטות תזמון כמו Cosine Decay.

קצב למידה ותזמוני אופטימיזציה

האופטימייזר הוא רק חלק מהתמונה; תזמון של קצב למידה חשוב לא פחות. קצב למידה קבוע כמעט אף פעם לא מתאים לכל שלבי האימון. אסטרטגיות נפוצות כוללות:

Warmup + Decay: התחלה בקצב למידה נמוך, עלייה הדרגתית ואז דעיכה.

Cosine Annealing: הפחתה הדרגתית של הקצב לפי עקומת קוסינוס, לרוב יחד עם Warmup.

AdamW ו־Lion מציגים ביצועים טובים עם תזמון של קצב הלמידה לפי פונקציית קוסינוס, בעוד ש־Adafactor משולב לעיתים עם תזמונים אדפטיביים לפי נורמות פרמטרים. בחירת תזמון נכון יכולה להשפיע משמעותית על מהירות ההתכנסות ועל הביצועים.

שיקולי יציבות והכללה

אופטימייזרים אדפטיביים כמו Adam ו־Lion מתכנסים מהר, אך עלולים להיקלע ל־**Minima Sharp** - נקודות במרחב הפרמטרים שממזערות מחיר אימון אך מכלילות גרוע. SGD עם מומנטום, על אף איטיותו, נוטה למצוא נקודות מינימום נמוכות יותר - מה שמוביל לדיוק טוב יותר.

בתנאים רועשים במיוחד, שיטות אדפטיביות (ובעיקר AdamW) נוטות להיות יציבות יותר. עם זאת, מעבר לאופטימייזרים פשוטים כמו SGD בשלב Fine-Tuning עשוי לשפר רובסטיות.

טבלת השוואה

אופטימייזר	זיכרון	קצב למידה אדפטיבי	מומנטום	שימוש אופייני
SGD	נמוך	לא	כן	משימות רובסטיות
AdamW	בינוני	כן	כן	רוב המודלים
Lion	נמוך	לא	סימן	ViTs, מעט זיכרון
Adafactor	נמוך מאוד	כן	לא	מודלים עצומים

טבלה 14.1: סיכום אופטימייזרים מודרניים

טיפים

עבור דאטהסטים רועשים או גרדיאנטים לא יציבים, AdamW הוא בחירה בטוחה ויעילה. עבור סביבות GPU מוגבלות - במיוחד באימון ViTs - Lion עשוי לספק יעילות גבוהה יותר. במודלים גדולים מאוד (למשל מעל מיליארד פרמטרים), Adafactor הופך חיוני בזכות יעילותו.

סיכום

האופטימייזרים המודרניים הם הרבה יותר מ־Gradient Descent פשוט. הם משלבים רעיונות של מומנטום, קצבי למידה אדפטיביים, יעילות זיכרון ורגולריזציה - כל אחד מהם מציע יתרונות בתרחיש שונה.

AdamW אומץ באופן נרחב בזכות איזון בין יציבות לביצועים; Lion מציע פשטות ומהירות במודלים ויזואליים תחת מגבלות משאבים; Adafactor מאפשר אימון בקנה מידה עצום; ו־SGD, על אף פשטותו, נותר יעיל בעיקר בשלבי Fine-Tuning.

שיעור 15

Explainability: להבין החלטות של מודלים

מבוא

ככל שמודלים של למידה עמוקה משמשים יותר ויותר בקבלת החלטות בעלות סיכון גבוה, היכולת להסביר את פעולתם הופכת להכרחית. בתחומים כמו בריאות, פיננסים, מערכות אוטונומיות וביטחון - כבר לא מספיק שמודל יהיה מדויק; עליו גם להיות מובן.

Explainability מתייחסת למגוון שיטות המספקות תובנה מדוע מודל ייצר פלט מסוים. לשיטות אלו מטרות רבות: הן מגבירות אמון משתמשים, עוזרות למפתחים לנפות התנהגויות בלתי צפויות, מבטיחות עמידה ברגולציה וחושפות סוגיות אתיות או של הוגנות. בסופו של דבר, מודל שלא ניתן להסביר - הוא מודל שיהיה קשה לסמוך עליו.

מדוע קשה להסביר רשתות עמוקות

מודלים ליניאריים ועצי החלטה מספקים שקיפות מובנית: ניתן לבחון ישירות מקדמים או כללי החלטה. לעומת זאת, רשתות נוירונים עמוקות פועלות באמצעות שכבות רבות של טרנספורמציות לא־ליניאריות, לרוב עם מיליוני פרמטרים. הייצוגים הפנימיים שלהן מבוזרים, שזורים ובדרך כלל בלתי ניתנים לפרשנות ללא ניתוח Post Hoc.

מורכבות זו מקשה לקבוע איזה מידע השפיע על החלטה מסוימת, או

לזהות מתי המודל מסתמך על קורלציות מקריות או משתנים מעורבים.

מקרה לדוגמה: גילוי הונאות בבנקאות

נניח בנק המפעיל מודל למידה עמוקה לגילוי עסקאות של הונאת אשראי. במהלך הבדיקות המודל מציג דיוק גבוה, אך צוות הרגולציה דורש שכל עסקה חשודה תלווה בהסבר. כשאנליסטים חוקרים מספר False Positives, הם מגלים תבנית מטרידה: הונאות רבות מרוכזות סביב עיר מסוימת.

באמצעות הסברי SHAP, הם מגלים שהתכונה "מיקום עסקה" תורמת רבות להחלטה - אך רק כאשר היא משולבת עם קטגוריות מסחר מסוימות. הדבר מוביל לביקורת עמוקה יותר, החושפת שהמודל למד לקשר פעילות "חשודה" לשילוב של התנהגויות לגיטימיות לחלוטין.

במקביל, אנליסטים מפעילים Grad-CAM על תת־מודול מבוסס CNN שמנתח קבלות סרוקות. Feature Maps מגלות שהמודל מתמקד לעיתים בטקסט מודגש - ולא בפרטי העסקה - בשל הטיות בדאטהסט האימון.

שתי שיטות ההסבר הללו, שיושמו ברמות שונות במערכת, סייעו לצוות לאמן מחדש את המודל, לשפר את תהליך הדאטה ולעמוד בדרישות הרגולטוריות לשקיפות.

Grad-CAM: הסבר חזותי ב־CNNs

ברשתות קונבולוציה, כלי סטנדרטי להסבר פעולת המודל הוא **Grad-CAM** (Gradient-weighted Class Activation Mapping). Grad-CAM מייצרת מפת חום המדגישה את האזורים המרחביים בתמונה שהשפיעו ביותר על החלטת המודל.

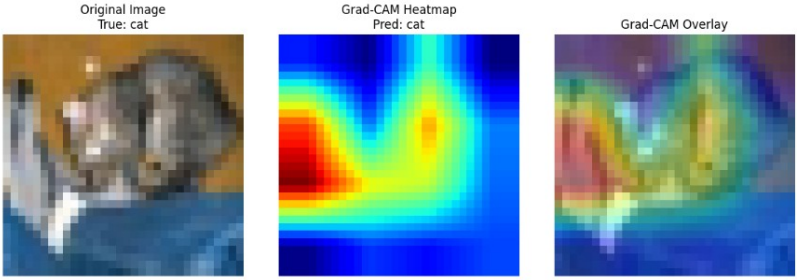
נסמן ב- y^c את ציון המודל (לפני Softmax) עבור מחלקה c ונסמן $A^k \in \mathbb{R}^{H\times W}$ כמפת הפיצ'רים ה־k מהשכבה הקונבולוציונית שנבחרה. החשיבות α_k^c מחושבת כך:

$$\alpha_k^c = \frac{1}{Z}\sum_{i=1}^{H}\sum_{j=1}^{W}\frac{\partial y^c}{\partial A_{i,j}^k} \tag{15.1}$$

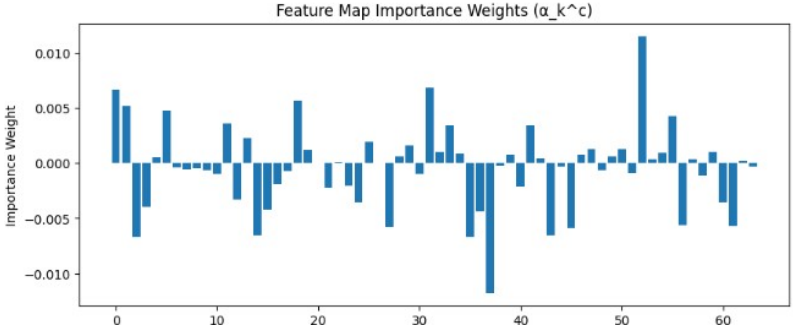
כאשר $Z = H \cdot W$ הוא מספר האלמנטים המרחביים במפת הפיצ'רים. משקלים אלו מייצגים עד כמה כל מפת פיצ'רים משפיעה על ציון המחלקה. מפת ה־Grad-CAM עצמה מחושבת כך:

$$L_c^{\text{Grad-CAM}} = \text{ReLU}\left(\sum_k \alpha_k^c A^k\right) \tag{15.2}$$

הפעלת ReLU מבטיחה שרק אזורים שתורמים ערכים חיוביים. מפת החום מוגדלת חזרה לגודל הקלט לצורך ויזואליזציה.



איור 15.1: דוגמה - המחשת ניבוי "חתול" ב־CIFAR-10. התמונה מציגה הסבר Grad-CAM עבור CNN שאומן על CIFAR-10. המודל סיווג נכון את התמונה כ־cat ו־Grad-CAM מדגיש את האזורים הרלוונטיים ביותר שהובילו לניבוי.



איור 15.2: החשיבות α_k^c עבור כל מפת פיצ'רים. כל מפת פיצ'רים לומדת תבנית ויזואלית מסוימת של ה־CNN (למשל טקסטורות, צורות, חלקים). משקלים חיוביים תומכים בסיווג "חתול"; שליליים סוננו על ידי ReLU.

חשיבות גלובלית: ניתוח מבוסס Permutation

במודלים טבלאיים, הפרשנות מתמקדת לעיתים בזיהוי אילו פיצ'רים חשובים באופן גלובלי. בעוד שמודלים מבוססי עצים (כמו XGBoost) מספקים מדדים פנימיים, גישה אינטואיטיבית ואגנוסטית־למודל היא **Permutation Importance**.

נניח f הוא מודל מאומן ו־$\mathcal{D}$ דאטהסט ולידציה. עבור כל פיצ'ר x_i, מחליפים את ערכיו אקראית ומודדים את שינוי הביצועים. פורמלית:

$$\Delta_i = M_0 - M_i \tag{15.3}$$

כאשר M_0 הוא המדד המקורי (למשל דיוק) ו־M_i המדד לאחר שינוי x_i. ערך גדול של Δ_i מצביע על כך שהפיצ'ר חיוני לביצועים.

SHAP: הסברים מקומיים מתמטיים

SHAP (SHapley Additive exPlanations) היא שיטה לכימות תרומת פיצ'רים לפלט המודל. היא מבוססת על ערכי Shapley מתורת המשחקים ומקיימת ארבעה אקסיומות: יעילות, סימטריה, דמה ואדיטיביות - המבטיחות ייחוס הוגן ועקבי.

נסמן F כקבוצת הפיצ'רים המלאה ו־$f(S)$ את ניבוי המודל באמצעות תת־קבוצה $S \subseteq F$. ערך ה־SHAP עבור פיצ'ר i מוגדר כך:

$$\phi_i = \sum_{S \subseteq F \setminus \{i\}} \frac{|S|!(|F| - |S| - 1)!}{|F|!} \left[f(S \cup \{i\}) - f(S) \right] \tag{15.4}$$

כאן:

- ϕ_i הוא ציון הייחוס של פיצ'ר i.
- $f(S)$ הוא פלט המודל הצפוי עם תת־קבוצת הפיצ'רים S.
- הפקטור הקומבינטורי משקלל את כל הצירופים האפשריים של הפיצ'רים.

בפועל, חישוב מדויק של משוואה זו יקר מאוד עבור פיצ'רים רבים. לכן משתמשים בקירובים כגון Kernel SHAP (אגנוסטי למודל) או Deep SHAP (לרשתות נוירונים). אלו משתמשים בהתפלגויות ידועות או דוגמאות ייחוס. למרות העלות, SHAP מספק הסברים יציבים ואינטואיטיביים ברמת הדוגמה הבודדת והפך לכלי מרכזי בתעשיות מפוקחות.

הסבר טקסט באמצעות Attention Weights

במודלים מבוססי Transformers כגון BERT או GPT, פרשנות מוטמעת כבר בארכיטקטורה באמצעות מנגנון ה־Self-Attention. כל טוקן מחושב כצירוף משוקלל של כל שאר הטוקנים, ליצירת ייצוג הקשרי.

נזכיר את נוסחת ה־Attention מהשיעורים הקודמים:

$$\text{Attention}(i,j) = \frac{\exp(q_i^\top k_j / \sqrt{d_k})}{\sum_{j'=1}^{T} \exp(q_i^\top k_{j'} / \sqrt{d_k})} \tag{15.5}$$

כאשר:

- $q_i, k_j \in \mathbb{R}^{d_k}$ הם וקטורי ה־Query וה־Key של טוקנים i ו־j.
- d_k הוא ממד מרחב ה־Key/Query.
- T הוא מספר הטוקנים ברצף.

משקלי ה־Attention יוצרים מטריצה $A \in \mathbb{R}^{T \times T}$ שניתן להציג כדי לגלות אילו מילים משפיעות זו על זו בכל שכבה וראש. למרות ש־Attention אינו מספק הסבר ישיר לניבוי, הוא חושף את אופן פעולת המודל בהבנת הקשר, במיוחד במשימות של תרגום או ניתוח תחבירי.

מגבלות ואתגרים

למרות תועלתן, לשיטות פרשנות מגבלות שונות: Grad-CAM מוגבל ברזולוציה ועלול לייצר מפות רועשות בשכבות עמוקות. SHAP, על אף היסודות המתמטיים,

רגיש לקורלציה בין פיצ'רים ויקר חישובית במימד גבוה. Attention Weights משקפים אסוציאציה ולא סיבתיות ויכולים להשתנות בין ראשים ושכבות.

הסברים גם עלולים להטעות: מודל עשוי לייחס ערכי SHAP גבוהים לפיצ'רים שהם רק "פרוקסים" למשתנים אחרים - עשויים להדגיש טוקנים תחביריים ללא רלוונטיות סמנטית.

לכן, Explainability צריך להיחשב עדשה דיאגנוסטית - לא מקור אמת מוחלט. שילוב עם ידע תחומי, בדיקות רובסטיות והערכה אנושית - חיוניים.

סיכום

Explainability הופך למידה עמוקה מקופסה שחורה למערכת הניתנת לביקורת ולפעולה. Grad-CAM מדגיש אזורים חזותיים משמעותיים ב־CNNs. SHAP מספק ייחוס מתמטי מבוסס תורת המשחקים לניבויים אינדיבידואליים. שיטות Permutation מספקות תובנות גלובליות פשוטות אך אפקטיביות. Attention Weights פותחים חלון להבנת אינטראקציות ברמת טוקן במודלים מבוססי Transformers.

לכל טכניקה יתרונות ומגבלות. הבחירה ביניהן תלויה בסוג הדאטה, המודל ומטרות הפרשנות.

שיעור 16

TensorFlow מול PyTorch

מבוא

לאחר שלמדנו כיצד לתכנן, לאמן ולממש רשתות נוירונים, עולה לעיתים שאלה פרקטית: באיזה פריימוורק להשתמש? שתי הספריות הפופולריות ביותר כיום הן TensorFlow ו־PyTorch. שתיהן בשימוש נרחב, מתוחזקות היטב ותומכות הן בסביבות מחקר והן בסביבות פרודקשן. הן חולקות הרבה מהפונקציונליות - אך נבדלות בפילוסופיה, בתחביר ובדפוסי שימוש אופייניים.

בשיעור זה נשווה בין שתי הספריות מבחינת מבנה, סגנון ואקוסיסטם. בנוסף נציג דוגמת סיווג עם Fashion MNIST כדי להמחיש את ההבדלים בקוד.

רקע היסטורי ואקוסיסטם

TensorFlow הוצג על ידי גוגל בשנת 2015 כפלטפורמה סקלאבילית מוכוונת פרודקשן. הוא הפך לברירת מחדל עבור יישומים ארגוניים רבים, בזכות כלים כמו TensorFlow Serving, TensorFlow Lite ו־TensorFlow.js.

PyTorch הושק על ידי פייסבוק בשנת 2016 וזכה במהירות לפופולריות בקהילת המחקר. התחביר הפייתוני הטבעי שלו, מודל ההרצה הדינמי והגמישות הפכו אותו לבחירה מועדפת לניסויים ולפרוטוטייפים מהירים.

עם הזמן, שתי הספריות התפתחו. TensorFlow 2 אימצה ברירת מחדל

של Eager Execution ו־PyTorch כוללת כיום כלים למימוש, כמו TorchServe ו־TorchScript. שתיהן תומכות ב־ONNX (Open Neural Network Exchange) ומשתלבות היטב עם כלי ML מודרניים.

מודלי הרצה: סטטי מול דינמי

היסטורית, TensorFlow דרשה מהמשתמש להגדיר גרף חישוב סטטי מראש, שהורץ בתוך Session. מודל זה הקל על אופטימיזציה אך צמצם גמישות.

לעומת זאת, PyTorch השתמשה מלכתחילה בחישוב דינמי: הקוד מורץ שורה אחר שורה כפי שנכתב בפייתון. זה מקל על דיבוג ובניית מודלים עם מנגנון בקרה מורכב.

ב־TensorFlow 2, ברירת המחדל היא Eager Execution. יחד עם זאת, ניתן להשתמש ב- @tf.function כדי להמיר פונקציות פייתון לייצוגי גרף - שילוב של גמישות וביצועים.

מקרה לדוגמה: סיווג Fashion MNIST

כעת נדגים מימוש עם שתי הספריות על משימת סיווג זהה: Fashion MNIST. הדאטהסט כולל תמונות גווני אפור (28×28 פיקסלים) של פריטי לבוש, מחולקות ל־10 מחלקות. המטרה: לממש רשת נוירונים פשוטה לסיווג.

מימוש ב־PyTorch

```
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

train_data = datasets.FashionMNIST('.', train=True, download=True, transform=
    transforms.ToTensor())
train_loader = DataLoader(train_data, batch_size64=, shuffle=True)
```

```
class Net(nn.Module:)
    def __init__(self:)
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )

    def forward(self, x:)
        return self.net(x)

model = Net()
optimizer = torch.optim.Adam(model.parameters(), lr)0.001=
loss_fn = nn.CrossEntropyLoss()

for x, y in train_loader:
    pred = model(x)
    loss = loss_fn(pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

מימוש ב־TensorFlow

באמצעות Keras, TensorFlow מציעה ממשק דקלרטיבי יותר. היא מפשטת את לולאת האימון ואת הלוגיקה הפנימית, מה שהופך אותה לאידיאלית לניסויים מהירים או למפתחים חדשים.

תחביר, מודולריות ורמת הפשטה

הקוד מעלה ממחיש את ההבדל הסגנוני המרכזי:

- PyTorch מדגישה שקיפות ומודולריות. המשתמש מגדיר במפורש את המודל, את חישוב המידע ברשת ואת לוגיקת האופטימיזציה.

- TensorFlow עם Keras מפשטת פרטים אלה, מאפשרת פיתוח מהיר יותר, אך דורשת קונפיגורציה עמוקה יותר במשימות מתקדמות.

קוד PyTorch מרגיש כמו פייתון טבעי, בעוד שקוד TensorFlow - במיוחד בהגדרת מודלים פונקציונליים או במצב גרף - מרגיש יותר כמו framework עם מוסכמות משלו.

אקוסיסטם ופרודקשן

בבחירת framework יש להתחשב לא רק בפיתוח אלא גם במימוש:

TensorFlow תומך ב:

- **TensorFlow Serving** למימוש ב־REST ו־gRPC
- **TensorFlow Lite** להרצה במובייל/אמבדד
- **TensorFlow.js** להרצה בדפדפן

PyTorch תומך ב:

- **TorchScript** לייצוא מודלים כגרפים סריאליים
- **TorchServe** לפריסת מודלים
- ייצוא ל־**ONNX**, המאפשר מימוש במסגרות אחרות

שתי הספריות תומכות באימון Mixed-Precision, האצה ב־GPU/TPU ואינטגרציה עם Hugging Face, Weights & Biases ו־MLFlow.

מתי להשתמש בכל אחת

הבחירה תלויה בהקשר. PyTorch מצוינת לניסויים מהירים, מחקר והוראה. היא מספקת שליטה מדויקת ודיבוג ברור. TensorFlow מצטיינת במימוש קצה־לקצה, מערכות פרודקשן ארוכות־טווח ופריסות מובייל/וב.

בזכות תמיכת ONNX, ניתן להעביר מודלים ביניהן. נפוץ יותר ויותר לבצע פרוטוטייפ ב־PyTorch ולהמיר ל־TensorFlow לפרודקשן - או להפך.

סיכום

גם TensorFlow וגם PyTorch הן ספריות מודרניות עוצמתיות. הן חולקות תכונות רבות ודומות בתכן. הבדלים מרכזיים נמצאים בתחביר, ברמת ההפשטה ובכלי האקוסיסטם. בחרו ב־PyTorch לשליטה, גמישות ומידול דינמי. בחרו ב־TensorFlow (Keras) לפיתוח מהיר ומימוש יעיל בפרודקשן.

שיעור 17

עבודה אפקטיבית עם Google Colab

מבוא

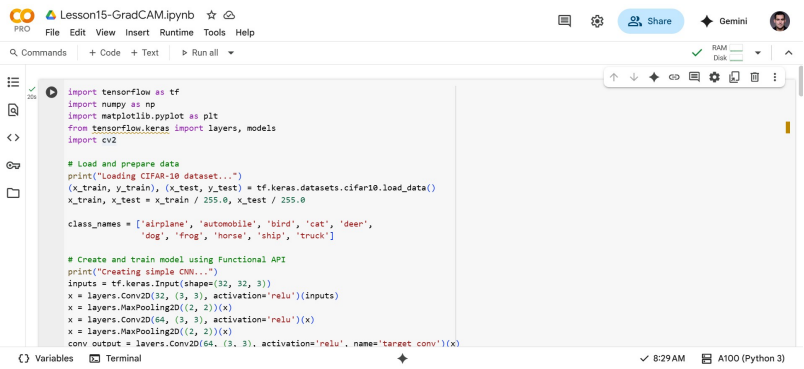
Google Colab הפך לנקודת התחלה עבור מפתחים, סטודנטים וחוקרים רבים בתחום הלמידה העמוקה - ובצדק. הוא חינמי, מבוסס ענן, תומך ב־GPU ומאפשר שיתוף ועבודה משותפת בקלות. אך כמו כל סביבה מבוססת ענן, שימוש יעיל בו דורש הבנה של מגבלותיו, קיצורי הדרך והיכולות הנסתרות שלו.

שיעור זה מציג את תהליך העבודה ב־Colab, כולל הגדרות חומרה, ניהול קבצים, טיפים להתקנות, אימון מודלים ופרקטיקות מומלצות להימנעות מניתוקים ולמקסום פרודוקטיביות.

תחילת עבודה בסביבת Colab

כאשר פותחים מחברת חדשה ב־Colab, עובדים על Jupyter Notebook מבוסס ענן הרץ על התשתית של גוגל. כל תא במחברת יכול להכיל קוד, הסברים ב־Markdown, או תוצאות. הקוד מורץ במכונה וירטואלית מבודדת ומשאבים כמו זיכרון, דיסק ושימוש ב־GPU מוצגים בפינה הימנית העליונה של הממשק.

ניתן לעבור בין בלוקי קוד לטקסט באמצעות הכפתורים "+ Code" ו־"+Text" בסרגל העליון.



איור 17.1: צילום מסך של Colab Google .

הפעלת מאיצי GPU או TPU

Colab מאפשר לבקש גישה ל־GPU או TPU, מה שמאיץ משמעותית את האימון.

1. גשו ל־Runtime > `Change runtime type`
2. בחרו GPU או TPU
3. לחצו Save

לאחר ההפעלה, ניתן לאמת את הגישה ל־GPU באמצעות:

```
import torch
print(torch.cuda.is_available())
```

או על ידי בדיקת ההתקן הפעיל:

```
!nvidia-smi
```

פלט נפוץ כולל התקנים כמו Tesla T4 או V100.

התקנת חבילות Python

Colab תומך בהתקנת חבילות ישירות באמצעות pip בתאי המחברת. לדוגמה:

```
!pip install transformers --upgrade --quiet
```

הדגל quiet– מצמצם לוגים ארוכים של ההתקנה.

חיבור ל־ Google Drive

כדי לשמור מודלים, פלטים או דאטהסטים בין סשנים, יש להגדיר את Google Drive:

```
from google.colab import drive
drive.mount('/content/drive')
```

הקבצים יהיו זמינים תחת /content/drive/MyDrive/. כך מבטיחים שהתוצאות לא יאבדו כאשר הסשן מתאפס.

העלאת קבצים ידנית

ניתן להעלות קבצים מהמחשב המקומי על ידי:

- גרירה אל חלון ה־"Files Pane"
- שימוש בפקודה:

```
from google.colab import files
files.upload()
```

קבצים אלו נשמרים זמנית במערכת הקבצים המקומית של המחברת.

שיתוף ועבודה משותפת

כל מחברת Colab ניתנת לשיתוף כמו מסמך Google Doc. לחצו על כפתור "Share" בפינה הימנית העליונה ובחרו הרשאות. ניתן גם לפתוח מחברות ישירות מ־GitHub.

אימון רשת נוירונים עם GPU

נאמן רשת פשוטה עם Fashion MNIST באמצעות PyTorch ו תמיכה ב־GPU:

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch import nn, optim
from tqdm import tqdm

device = 'cuda' if torch.cuda.is_available() else 'cpu'

train_data = torchvision.datasets.FashionMNIST)root='.', train=True, download=
    True,
                        transform=transforms.ToTensor())
train_loader = torch.utils.data.DataLoader)train_data, batch_size64=, shuffle=True
    )

model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
).to(device)

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

for epoch in range(3):
    total_loss = 0
    for x, y in tqdm(rain_loader:)
        x, y = x.to(device), y.to(device)
        pred = model(x)
        loss = loss_fn(pred, y)
```

```
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss:.3f}")
```

ניטור עם TensorBoard

כדי לנטר את האימון באופן ויזואלי, Colab תומך ב־TensorBoard. תחילה יש לטעון את ההרחבה:

```
%load_ext tensorboard
%tensorboard --logdir runs
```

בתוך לולאת האימון, יש להגדיר את המדדים באמצעות:

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()

# האימון במהלך
writer.add_scalar('Loss/train', loss.item(), step)
```

טיפים מתקדמים

- שמירת Model Checkpoints:

```
torch.save(model.state_dict(), 'model.pth')
```

- בדיקת שימוש בדיסק:

```
!du -sh /content
```

- הצגת קבצים:

```
!ls -lh
```

- מדידת זמן ריצה של תא:

```
%%time
```

- בדיקת שימוש בזיכרון:

```
!free -h
```

סיכום

Google Colab היא פלטפורמה נוחה, נגישה עם שימוש ב־GPU ללמידה עמוקה. עם הגדרת עבודה נכונה, הוא הופך ליותר מלוח ניסויים - היא הופכת למעבדה מלאה בענן. למדו את הקיצורים הקיימים בסביבה לעבודבה יעילה, נצלו בחכמה את ה־Drive, נטרו את האימון באופן ויזואלי ועבדו במבנה מודולרי.

שיעור 18

Mixed Precision Training

מבוא

כעת ראינו כיצד בונים, מאמנים, מאפטמים ומעריכים רשתות עמוקות. אך יש שכבה אחרונה שמחברת את כל אלה - ממד נסתר שבו זמן, זיכרון ודיוק מתלכדים.

Mixed Precision Training אינו אופטימייזר חדש, לא טריק רגולריזציה ולא פונקציית מחיר חדשה. זהו שינוי יסודי באופן שבו אנו מייצגים ומבצעים חישובים בתוך הרשתות - ואם משתמשים בו נכון, הוא יכול להביא לאימון מהיר יותר, שימוש מופחת בזיכרון ולפעמים אפילו להכללה טובה יותר.

מה המשמעות של Precision בפועל

רוב הרשתות הנוירוניות כיום מאומנות באמצעות מספרים בפורמט נקודה צפה של 32 ביט (FP32). זהו הפורמט המקובל לאחסון משקולות, חישוב אקטיבציות ומעקב אחרי גרדיאנטים. הוא מציע יציבות נומרית גבוהה, אך במחיר: כל מספר צורך 32 ביט וכל פעולה משתמשת בנתיב חישוב יקר יחסית.

GPUs מודרניים, במיוחד מסדרות Turing ו־Ampere של NVIDIA, תומכים באלטרנטיבה: מספרים בפורמט 16 ביט נקודה צפה (FP16). אלו קטנים יותר, מהירים יותר ויעילים יותר בזיכרון - אך מקריבים כמות משמעותית של דיוק

נומרי. ההבדל אינו רק באחסון. FP32 משתמש ב־8 ביט לאקספוננט ו־23 למנטיסה, בעוד FP16 משתמש ב־5 ביט לאקספוננט ו־10 למנטיסה. טווח מצומצם זה עלול להוביל לאי־יציבות אם לא מטופל כראוי.

למרות זאת, שימוש ב־FP16 יכול להכפיל את רוחב הפס של הזיכרון ולהפחית משמעותית את זמן האימון. האתגר הוא להחליט היכן ניתן להשתמש בו באופן בטוח - והיכן לא.

הגישה של Mixed Precision

במקום לבחור בין FP32 ל־FP16, צינורות אימון מודרניים משלבים את שניהם. זהו הרעיון של Mixed Precision. כברירת מחדל, חישובים שאינם רגישים לדיוק נמוך מבוצעים ב־FP16. חישובים רגישים - במיוחד כאלה שכוללים ערכי Loss, גרדיאנטים קטנים או עדכוני משקולות - מתבצעים בדיוק מלא FP32.

הפרדה זו מאפשרת לנצל את היתרונות של מהירות וזיכרון ב־FP16, מבלי לפגוע ביציבות המודל. המעבר מנוהל אוטומטית על ידי פריימוורקים כמו PyTorch ו־TensorFlow, באמצעות כלים כגון AMP - Automatic Mixed Precision.

AMP ב־ PyTorch: דוגמה פשוטה

PyTorch הופך את השימוש ב־Mixed Precision לפשוט במיוחד. שני רכיבים מרכזיים torch.cuda.amp: autocast, שמבצע המרות אוטומטיות ל־FP16 היכן שניתן ו־GradScaler, שמגן מפני Gradient Underflow על ידי סקיילינג של ה־Loss לפני Backpropagation ו־Unscaling לפני צעד האופטימיזציה.

דוגמה מלאה לאימון ResNet18 על CIFAR-10:

```
import torch
from torch import nn, optim
from torchvision.models import resnet18
from torchvision.datasets import CIFAR10
from torchvision.transforms import ToTensor
```

```
from torch.utils.data import DataLoader
from torch.cuda.amp import autocast, GradScaler
from tqdm import tqdm

device = 'cuda' if torch.cuda.is_available() else 'cpu'

train_data = CIFAR10(root='.', train=True, download=True, transform=ToTensor
    ())
train_loader = DataLoader)train_data, batch_size64=, shuffle=True(

model = resnet18(num_classes.)10=to(device)
optimizer = optim.Adam)model.parameters()(
loss_fn = nn.CrossEntropyLoss()
scaler = GradScaler()

for epoch in range(3):
    total_loss = 0
    for x, y in tqdm(train_loader:)
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        with autocast():
            pred = model(x)
            loss = loss_fn(pred, y)
        scaler.scale(loss.)backward()
        scaler.step(optimizer)
        scaler.update()
        total_loss =+ loss.item()
    print(f"Epoch {epoch}1+, Loss: {total_loss3.:f}")
```

Listing 18.1: אימון עם AMP: ResNet18 על CIFAR-10

לולאת האימון בגרסה זו דורשת רק שלושה שינויים: עטיפת ה־Forward ב־autocast, Pass החלפת loss.backward() בקריאה ל־scaler.scale(loss).backward() ושימוש ב־scaler.step במקום צעד האופטימיזציה הסטנדרטי. התאמות קטנות

אלו יכולות להביא להאצות דרמטיות - במיוחד באימון מודלים גדולים או בשימוש בבאצ'ים גדולים.

איך לדעת אם זה מועיל

כדי למדוד את התועלת האמיתית של Mixed Precision, התחילו במעקב אחרי זמן ריצה. ניתן להשתמש ב־%%time בתוך תא אימון או להשוות אפוקים. לגבי זיכרון, הריצו nvidia-smi! בתא נפרד וצפו בירידה בצריכה. אחד היתרונות המידיים הוא האפשרות להגדיל את גודל הבאצ' מבלי להיתקל במגבלות זיכרון - לעיתים להכפיל או לשלש את ה־Throughput.

מתי זה עובד ומתי להיזהר

Mixed Precision Training עובד היטב ברוב התרחישים - אך לא בכולם. מודלים עמוקים או רחבים מאוד, כמו Transformers, מרוויחים במיוחד. גם משימות עם קלטים גדולים, כמו יצירת תמונות או עיבוד רצפים. במקרים אלו, AMP צריכה להיחשב כברירת מחדל.

עם זאת, יש חריגים: ב־GPU ללא Tensor Cores - כמו כרטיסי Kepler או Maxwell ישנים - ייתכן שלא תראו שיפור ואף האטה. במשימות שדורשות גילוי אנומליות עדינות, כמו אבחונים רפואיים או סיווג אירועים נדירים - ירידת הדיוק הנומרי **עלולה לפגוע** ברגישות המודל. לבסוף, יש מקרים חריגים שבהם נוצרים NaNs באימון עקב Underflow או Exploding Gradients. במקרה כזה ודאו שאתם משתמשים ב־GradScaler. אם הבעיה נמשכת, הורידו את קצב הלמידה או חזרו זמנית ל־FP32.

ברירת מחדל חדשה ב־2025

נכון ל־2025, אקוסיסטם הלמידה העמוקה מניח שימוש ב־Mixed Precision. סביבות כמו PyTorch Lightning ו־TensorFlow כבר מגדירות AMP כברירת מחדל. מרבית המחברות ב־Kaggle משתמשות בזה באופן מובנה. גם

ב־Hugging Face הרבה מסקריפטי האימון מפעילים זאת כברירת מחדל. אלא אם אתם במחקר שדורש במפורש דיוק נומרי גבוה - אין עוד סיבה לא להשתמש ב־AMP.

אם אתם מאמנים מודל גדול, משתמשים ב־GPU מודרני, או עובדים עם באצ'ים מעל 128 - עליכם להשתמש ב־Mixed Precision. היתרונות: התכנסות מהירה יותר ופחות צווארי בקבוק בזיכרון.

סיכום

Mixed Precision Training היא אחת מהטכניקות הנדירות בלמידה עמוקה שמשפרות גם ביצועים וגם יעילות - מבלי לדרוש שינוי ארכיטקטוני. היא מאפשרת למודלים לרוץ מהר יותר ולצרוך פחות זיכרון ועדיין להגיע לאותן תוצאות (או אפילו טובות יותר). בזכות כלים כמו AMP, שילוב בתהליך האימון הפך לטריוויאלי.

הרעיון הבסיסי פשוט: חשבו מהר היכן שניתן וחשבו בזהירות היכן שחובה. כמו בכל הנדסה טובה - מדובר על איזון. בעידן של מודלים גדולים ותקציבים מוגבלים, האיזון הזה יכול לעשות את כל ההבדל.

שיעור 19

Transfer Learning ו־Fine-Tuning

מבוא

אימון מודל למידה עמוקה מאפס דורש לרוב דאטהסטים עצומים, משאבים חישוביים משמעותיים וכיול נרחב. בפועל, לעיתים קרובות יעיל יותר לא להתחיל מאפס, אלא ממודל מאומן מראש - מודל שכבר למד ייצוגים שימושיים מדאטה גדול יותר. גישה זו נקראת **Transfer Learning**.

Transfer Learning מנצל את יכולות חילוץ הפיצ'רים של מודל שאומן על דומיין כללי (למשל ImageNet או Wikipedia) ומתאים אותו למשימה ייעודית (למשל סיווג תמונות רפואיות או ניתוח סנטימנט). התהליך של התאמת המודל המאומן מראש למשימה החדשה נקרא **Fine-Tuning** .

בשיעור זה נציג את המוטיבציה, האסטרטגיות והיישום של Transfer Learning ו־Fine-Tuning, עם דוגמאות מתחום הראייה הממוחשבת (CV) ו־NLP .

מהו Transfer Learning?

בלמידה מונחית קלאסית, אנו מאמנים מודל ממשקולות מאופסות אקראית בעזרת דוגמאות מתויגות. ב־Transfer Learning, אנו מתחילים ממודל שמשקולותיו כבר אומנו על דאטהסט רחב ודומה.

ברשתות עמוקות, השכבות המוקדמות נוטות ללמוד פיצ'רים כלליים

(קצוות, טקסטורות, דפוסים מקומיים), בעוד השכבות המאוחרות לומדות ייצוגים ספציפיים לדומיין. Transfer Learning מניח שייצוגים ברמה נמוכה אלה ניתנים לשימוש חוזר בין משימות שונות.

זוגמה: רשת קונבולוציה (CNN) שאומנה לסיווג אובייקטים בתמונות טבע (למשל כלבים, משאיות) יכולה להיות מותאמת לסיווג תמונות מוצרים או צילומי רנטגן באמצעות שימוש חוזר בשכבות המוקדמות ועדכון רק של שכבות הסיווג הסופיות.

מודלים מאומנים נפוצים

- **ראייה ממוחשבת (CV)**: Transformers Vision EfficientNet, MobileNet, ResNet, (ViT) - לרוב אומנו על ImageNet.
- **עיבוד שפה טבעית (NLP)**: DistilBERT GPT, RoBERTa, BERT, - לרוב אומנו על קורפוסי טקסט גדולים.
- **אודיו / דיבור**: Whisper wav2vec, - אומנו על גלי קול או ספקטרוגרמות.

מודלים מאומנים זמינים ברפוזיטוריז ציבוריים כמו Hugging Face Model Hub או Torchvision וניתן לטעון אותם בשורת קוד אחת.

שלוש אסטרטגיות Fine-Tuning

בדרך כלל מיישמים Transfer Learning באחת משלוש דרכים:

1. **Feature Extraction (Fixed Base)**: מקפיאים את כל השכבות המאומנות מראש ומאמנים רק שכבת פלט חדשה למשימה.
2. **Partial Fine-Tuning**: מקפיאים שכבות מוקדמות ומבצעים אימון לשכבות המאוחרות ולראש הפלט.
3. **Full Fine-Tuning**: מאפשרים לכל המשקולות להתעדכן, לרוב עם קצב למידה נמוך משמעותית.

כל גישה מאזנת בין גמישות ליציבות. Full Fine-Tuning עשוי להניב ביצועים גבוהים יותר במשימות ייחודיות לדומיין, אך דורש רגולריזציה קפדנית.

דוגמה: Fine-Tuning ל־ResNet על MNIST Fashion

נניח שאנו רוצים לסווג תמונות בגווני אפור של בגדים מתוך Fashion MNIST באמצעות Transfer Learning.

שלב 1: טעינת מודל מאומן מראש

```
from torchvision.models import resnet18
model = resnet18(pretrained=True)
```

שלב 2: הקפאת כל השכבות

```
for param in model.parameters():
    param.requires_grad = False
```

שלב 3: החלפת שכבת הפלט

```
import torch.nn as nn
model.fc = nn.Linear(512, 10)
```

שלב 4: אימון ראש הפלט

נשתמש בקצב למידה קטן (למשל 10^{-6}) ונאמן את שכבת הסיווג החדשה. ניתן בהמשך לשחרר בהדרגה שכבות נוספות.

שיקולים מתמטיים

במהלך Fine-Tuning, חשוב להשתמש בקצב למידה η מתאים. נסמן θ כוקטור הפרמטרים של המודל המאומן מראש ו־$\mathcal{L}$ כפונקציית המחיר למשימה החדשה. צעד העדכון הוא:

$$\theta_{k+1} = \theta_k - \eta \cdot \nabla_\theta L(\theta_k) \tag{19.1}$$

בדרך כלל מיישמים קצב למידה קטן יותר לשכבות המאומנות מראש וקצב גבוה יותר לשכבות החדשות. כך נמנע Catastrophic Forgetting של הייצוגים שנלמדו.

Fine-Tuning ב־NLP

מודלים לשפה טבעית כמו BERT מותאמים בגישה דומה. למשל:

```
from transformers import BertForSequenceClassification

model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", num_labels =2 )
```

פקודה זו טוענת את BERT עם ראש סיווג מותאם לניתוח סנטימנט בינארי. תהליך ה־Fine-Tuning מתבצע באמצעות Backpropagation רגיל על טקסט מתויג.

מתי לא להשתמש ב־ Transfer Learning

למרות עוצמתו, ל־Transfer Learning מגבלות:

- כאשר דומיין המקור והדומיין היעד שונים מאוד (למשל כלבים ← סריקות רפואיות).
- כאשר המודל המאומן מראש מכניס הטיות שאינן רלוונטיות למשימה החדשה.

- כאשר יש מספיק דאטה ייעודי לדומיין ומשאבים חישוביים כדי לאמן מאפס.

תמיד יש להשוות מול בסיס של מודל מאומן מאפס.

טיפים

- התחילו תמיד עם `requires_grad = False` ושחררו בהדרגה שכבות נוספות.
- השתמשו בקצב למידה נמוך לשכבות המאומנות מראש.
- עקבו אחרי ביצועי הולידציה כדי לזהות Overfitting.
- במודלים גדולים השתמשו בקצבי למידה שונים לשכבות (Layer-wise LR Scheduling).

סיכום

Transfer Learning מאפשר לנו לבנות מודלים מדויקים במהירות, עם פחות דאטה ופחות משאבים. הוא פועל על ידי שימוש חוזר בייצוגים כלליים ממודל בסיס שאומן על משימה קרובה והתאמתם באמצעות Fine-Tuning. גישה זו היא כיום סטנדרט כמעט בכל תת־תחום של למידה עמוקה.

בין אם מדובר בסיווג תמונות או בניתוח טקסט, Transfer Learning משנה את תהליך פיתוח המודל מאימון להתאמה - מאיץ את הפיתוח ומשפר את ההכללה.

שיעור 20

שמירה, טעינה ו־Versioning של מודלים

מבוא

לאחר השקעת שעות - או ימים - באימון מודל למידה עמוקה, הדבר האחרון שתרצו הוא לאבד את התוצאה. דיוק גבוה אינו אומר דבר אם לא ניתן לעשות שימוש חוזר במודל, לשחזרו, או להטמיעו.

שיעור זה עוסק בפרקטיקה מקצועית של שמירה, טעינה ו־Versioning של מודלים באופן שתומך בשיתוף פעולה, שחזוריות ותחזוקה ארוכת־טווח. נבחן את ההבדלים בין פורמטים כמו TorchScript, SavedModel ו־ONNX ונראה דוגמה מעשית עם CNN שאומן על Fashion MNIST.

חשיבות שמירת מודלים

שמירת מודל אינה רק צורך טכני - זהו עניין הנדסי. אם אינכם שומרים את הפרמטרים שאומנו, מבנה המודל וסביבת העבודה שבה נוצר - למעשה איבדתם את הניסוי. גם תוצאה של 94% דיוק חסרת משמעות אם לא ניתן לשחזרה.

בעולם האמיתי, אי שמירת מודלים ותיעוד קונפיגורציות עלול להוביל לאיבוד דאטה, בזבוז זמן ופגיעה בעבודת המשך. שמירה נכונה היא התחייבות לעצמכם בעתיד - או לאדם הבא שיעבוד על הקוד שלכם.

שלושת פורמטי השמירה המרכזיים

באקוסיסטם של למידה עמוקה ישנם שלושה פורמטים עיקריים לייצוא ושמירת מודלים מאומנים.

TorchScript (PyTorch)

TorchScript הוא פתרון PyTorch לייצוא מודלים לפורמט עצמאי. הוא מקמפל מודל לגרף חישוב סטטי, המאפשר הרצה מחוץ להרצת פייתון - למשל באפליקציות C++ או במובייל.

```
traced = torch.jit.trace(model, example_input)
traced.save("model.pt")
```

Listing 20.1: שמירה `TorchScript`

לטעינה ושימוש בinference:

```
model = torch.jit.load("model.pt")
model.eval()
```

Listing 20.2: טעינה `TorchScript`

הפורמט קומפקטי, מהיר וניתן להעברה - אידיאלי לאינטגרציה בפרודקשן עם תלות מינימלית.

SavedModel (TensorFlow)

ב־TensorFlow, פורמט ברירת המחדל הוא SavedModel. הוא שומר לא רק את המשקולות וגרף החישוב, אלא גם חתימות פונקציה ומטא־דאטה. השמירה פשוטה:

```
model.save("my_model")
```

Listing 20.3: שמירה `TensorFlow`

והטעינה פשוטה באותה מידה:

```
loaded = tf.keras.models.load_model("my_model")
```

Listing 20.4: טעינת SavedModel

פורמט זה מתאים במיוחד למימוש דרך TensorFlow Serving, TensorFlow Lite, או Google Cloud AI Platform.

ONNX (Open Neural Network Exchange)

ONNX הוא פורמט פתוח חוצה־פריימוורקים. מודל שאומן ב־PyTorch יכול להיות מיוצא ל־ONNX ולאחר מכן להיות ממומש בפריימוורקים שונים או שפות אחרות.

```
torch.onnx.export(model, dummy_input, "model.onnx",
                  input_names=["input"], output_names=["output"])
```

Listing 20.5: ייצוא מ־PyTorch ל־ONNX

ONNX אידיאלי לפריסת מודלים בסביבות שאינן קשורות לספרייה מסוימת, כגון Java, C++, או REST APIs המבוססים על ONNX Runtime או Triton Inference Server.

דוגמה: CNN על Fashion MNIST

נבחן דוגמה קונקרטית: נגדיר ונאמן רשת קונבולוציה קטנה לסיווג תמונות ונשמור אותה בשלושה פורמטים: משקולות גולמיות, TorchScript ו־ONNX.

```
import torch
import torch.nn as nn

class Net(nn.Module:)
    def __init__(self) :
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding= 1),
```

```
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(16 * 28 * 28, 10)
        )
    def forward(self, x) :
        return self.net(x)

model = Net()
```

Listing 20.6: CNN ב־ PyTorch

ראשית, ניתן לשמור רק את המשקולות:

```
torch.save(model.state_dict(), "cnn_weights.pth")

model.load_state_dict(torch.load)"cnn_weights.pth"))
model.eval()
```

Listing 20.7: שמירה+טעינה state dict

לאחר מכן, למימוש מחוץ ל־PyTorch:

```
example_input = torch.randn(1, 1, 28, )28
traced = torch.jit.trace(model, example_input)
traced.save("cnn_scripted.pt")
```

Listing 20.8: שמירה TorchScript

ולבסוף, לייצוא ל־ONNX:

```
dummy_input = torch.randn(1, 1, 28, )28
torch.onnx.export)model, dummy_input, "cnn.onnx",
                  input_names[="input"], output_names[="output")]
```

Listing 20.9: ייצוא ONNX ל־

בחירת הפורמט המתאים

הפורמט המתאים תלוי בהקשר המימוש, באקוסיסטם ובמטרות. לכל אפשרות יתרונות ייחודיים, כפי שמסוכם בטבלה:

פורמט	שימוש עיקרי	יתרונות
TorchScript	אינפרנס מהיר, אינטגרציה ל־C++/מובייל	קל משקל, נייד, רץ ללא פייתון
SavedModel	Serving TensorFlow או TFLite	כולל גרף, משקולות וחתימות; ידידותי לענן
ONNX	מעבר בין פריימוורקים	פורמט פתוח, נתמך ע"י רנטיימים רבים (Triton, ONNX Runtime, Optimum)

טבלה 20.1: השוואת פורמטי ייצוא מודלים

Versioning

שמירת מודל היא רק חלק מהסיפור. פרויקטים רציניים דורשים גם Versioning. כל מודל שנשמר צריך להישמר עם שם גרסה ברור, למשל model_v2.1.onnx. בנוסף, מומלץ לכלול קובץ לוג או מטא־דאטה נלווה, המכיל: תאריך אימון, תיאור קצר של השינויים, היפר־פרמטרים מרכזיים ומדדי ביצוע סופיים על דאטהסט הטסט. מידע זה מבטיח שאתם - או כל אדם אחר - תוכלו לשחזר את התוצאות גם חודשים מאוחר יותר. שקלו לשלב כלים כמו Git, MLflow, או DVC לניהול ניסויים וארטיפקטים של מודלים לאורך זמן ובין מכונות.

סיכום

שמירת מודל אינה סוף האימון - אלא תחילת חייו בעולם האמיתי. בין אם מדובר במימוש לפרודקשן, שיתוף עם הצוות, או חזרה לניסוי ישן - סריאליזציה נכונה ו־Versioning חיוניים.

פורמטים כמו TorchScript, SavedModel ו־ONNX מאפשרים להעביר מודלים בין שפות, רנטיימים ותשתיות. אך אף פורמט אינו יכול לשמר את הכוונה או ההקשר - זו כבר אחריות שלכם.

שיעור 21

מימוש בסיסי

מבוא

לאחר אימון ושמירת מודל למידה עמוקה, השלב החיוני הבא הוא להפוך אותו לשימושי - לא רק מתוך המחברת שלכם, אלא גם על ידי מערכות אחרות. זהו הגשר בין מודלינג לפרודקשן. כך מסווג תמונות הופך לשירות בזמן אמת, כך מנוע המלצות מניע אפליקציית ווב וכך למידה עמוקה הופכת לחלק ממוצר עובד.

בשיעור זה נראה כיצד לפרוס מודל סיווג תמונות מאומן כ־RESTful API באמצעות FastAPI ו־TorchScript. המטרה: לקבל תמונה דרך HTTP, להחזיר תחזית בפורמט JSON - נקי, מהיר ומוכן לפרודקשן.

ממודל לשירות

בסביבת פרודקשן לא מריצים מחדש סקריפטי אימון או מחברות כדי לקבל תחזיות. במקום זאת, עוטפים את המודל בשירות מתמשך - כזה שמאזין לבקשות נכנסות, מעבד אותן ומחזיר תחזיות. ארכיטקטורה זו מאפשרת לשלב את המודל באתרי אינטרנט, אפליקציות מובייל, דאשבורדים, מערכות IoT, או כל ממשק תוכנה אחר.

API של מודל מקבל קלט (למשל תמונה), מבצע פרה־פרוססינג, מפעיל אינפרנס באמצעות מודל טעון מראש ומחזיר את התוצאה במבנה מסודר

כמו JSON.

FastAPI : שירות אינפרנס עם בהירות

FastAPI הוא פריימוורק פייתון לבניית APIs עתירי ביצועים. הוא משתלב טבעית עם PyTorch, PIL, NumPy ותומך בעיבוד אסינכרוני ולידציה אוטומטית של בקשות ותיעוד אינטראקטיבי בזמן אמת דרך Swagger.

העיצוב שלו הופך אותו מתאים במיוחד לשירותי אינפרנס: קל משקל, מהיר לעלות ופשוט לבדיקה.

Case Use : סיווג תמונות עם TorchScript

נניח שאימנו רשת קונבולוציה על Fashion MNIST, ייצאנו אותה כ־cnn_scripted.pt ואנו רוצים לחשוף אותה כ־REST API שמקבלת תמונה ומחזירה את המחלקה החזויה. להלן המימוש המלא:

```
from fastapi import FastAPI, File, UploadFile
from fastapi.responses import JSONResponse
from PIL import Image
import torch
import torchvision.transforms as transforms
from io import BytesIO

# Device configuration
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Class labels for Fashion MNIST
LABELS = {
    :0 "T-shirt/top", :1 "Trouser", :2 "Pullover",
    :3 "Dress",       :4 "Coat",    :5 "Sandal",
    :6 "Shirt",       :7 "Sneaker", :8 "Bag", :9 "Ankle boot"
}
```

```
# Load model once on startup
model = torch.jit.load("cnn_scripted.pt", map_location=device)
model.eval()

# Define image preprocessing
transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((28, 28)),
    transforms.ToTensor()
)]

app = FastAPI()

@app.post("/predict/")
async def predict(file: UploadFile = File(...):)
    try:
        image = Image.open(BytesIO(await file.read()((.convert)"RGB"(
        x = transform(image).unsqueeze(0).to(device)

        with torch.no_grad():
            output = model(x)
            probs = torch.softmax(output, dim1)=
            pred_idx = probs.argmax(dim.)1=item()
            confidence = probs.max(dim.)1=item()
            label = LABELS[pred_idx]

        return JSONResponse(content={
            "prediction_index": pred_idx,
            "label": label,
            "confidence": round(confidence, 4)
        )}

    except Exception as e:
        return JSONResponse(status_code400=, content{="error": str)e()}
```

Listing 21.1: FastAPI+TorchScript

השירות טוען את המודל פעם אחת לזיכרון, מבצע פרה־פרוססינג לתמונה שהועלתה כך שתתאים לקלט המודל ומחזיר את האינדקס של המחלקה עם תווית וביטחון התחזית.

הרצת השירות מקומית

כדי להריץ את השירות מקומית, שמרו את הקוד בקובץ api.py ואז הריצו:

```
uvicorn api:app --host 0.0.0.0 --port 8000 --reload
```

Listing 21.2: הרצת Uvicorn עם `FastAPI`

לאחר ההרצה, בקרו בכתובת:

http://127.0.0.1:8000/docs

FastAPI תיצור באופן אוטומטי ממשק web שבו תוכלו להעלות תמונות, להריץ תחזיות ולבדוק תוצאות. זו אחת הסיבות לפופולריות שלו בקרב מתכנתים בתחום - בדיקת מודל הופכת פשוטה כמו שימוש באתר.

מעבר לענן

לאחר שהשירות פועל מקומית, ניתן לממש אותו בכל סביבה מודרנית. אפשרויות נפוצות כוללות:

- אחסון בפלטפורמות כמו Render או Hugging Face
- הרצה ב־Colab עם ngrok לחשיפת פורט מקומי
- מימוש במכונה וירטואלית ב־GCP, AWS, או Azure, לרוב בתוך קונטיינר Docker

בכל המקרים, המבנה נשאר זהה, אך נקודת הכניסה היא כעת Endpoint ציבורי שמוכן להחזיר תחזיות.

טיפים ליציבות וביצועים

כדי להבטיח שה־API שלכם יתנהג בעקביות וביעילות בפרודקשן:

- הקפידו להגדיר model.eval() לפני אינפרנס כדי להבטיח התנהגות דטרמיניסטית.

- עטפו את בלוק התחזית ב־torch.no_grad() כדי לחסוך בזיכרון ולנטרל מעקב גרדיאנטים.

- הימנעו מטעינת המודל בכל בקשה - טעינו אותו פעם אחת כשהשרת עולה.

- השתמשו ב־map_location=torch.device("cpu") בטעינת מודלים בסביבות ללא GPU. אחרת, ייתכנו שגיאות אם המודל נשמר על GPU.

- לשיפור מהירות תגובה, שמרו את טרנספורמציות התמונות קלות ובצעו עיבוד באותו התקן שבו רץ המודל.

- אם אתם מתכננים לטפל במספר בקשות לשנייה, בטלו את דגל ה־reload של FastAPI והשתמשו בשרת פרודקשן כמו Gunicorn או Uvicorn עם מספר עובדים. במערכות מתקדמות ניתן לממש Batch Inference או תורים אסינכרוניים באמצעות Celery או Ray Serve.

סיכום

מימוש מודל למידה עמוקה הוא הרגע שבו המודל הופך למערכת. REST API פשוט, המבוסס על FastAPI ו־TorchScript, יכול להפוך סקריפט מקומי לשירות רב־שימושי, ניתן לבדיקה ונגיש.

לא נדרש לכך הרבה קוד. אך כן נדרש תכנון ברור של ממשקים, התקנים ויציבות. מרגע שהמודל מומש, הוא כבר לא רק קובץ - אלא חלק ממוצר.

שיעור 22

התמחויות

מבוא

אם הגעתם לנקודה זו, כבר יש לכם בסיס טוב בלמידה עמוקה. אתם מבינים כיצד רשתות נוירונים נבנות, מתאמנות, מאופטמות ומיושמות. אתם יודעים מה זה אומר לספק תחזיות בזמן אמת וכיצד לנהל מודלים באופן מבוקר ובזהירות.

אך למידה עמוקה איננה תחום יחיד - היא אקוסיסטם של תחומים מתמחים. כל תחום מגיע עם האתגרים שלו, ארכיטקטורות מודלים ייחודיות ומוסכמות משלו. בשיעור זה נמפה את ארבעת הכיוונים הנפוצים ביותר להתמחות מקצועית: ראייה ממוחשבת, עיבוד שפה טבעית, למידה טבלאית וניתוח סדרות זמן.

מדוע התמחות חשובה

אין ארכיטקטורת מודל אחת שמתאימה לכל הבעיות. מודל שמצטיין בזיהוי תמונות עלול להיכשל לחלוטין כאשר מיושם על טקסט. שיטה שפותחה לדאטה טבלאי של CRM עשויה להתקשות ללמוד תלות טמפורלית בתחזיות פיננסיות. התמחות בלמידה עמוקה איננה בחירת אלגוריתם מועדף - אלא התאמת הכלים למבנה הדאטה ולטבע המשימה. הבנת המיפוי הזה היא המפתח לעבודה יעילה בלמידה עמוקה יישומית.

ראייה ממוחשבת (Computer Vision)

דאטה ויזואלי מציב אתגרים ייחודיים. בניגוד לוקטורים שטוחים, תמונות מכילות מבנה מרחבי, תלות מקומית ולעיתים קבצים גדולים. המודלים חייבים לחלץ פיצ'רים באופן היררכי תוך שמירה על יחסים מרחביים.

רשתות ראייה מוקדמות נשענות על שכבות קונבולוציה, שסורקות אחר דפוסים באזורים מקומיים. ארכיטקטורות כמו CNNs הן יסוד למשימות כמו סיווג, זיהוי וסיגמנטציה. לזיהוי מיקום אובייקטים, ארכיטקטורות כמו YOLO ו־DETR מציגות מנגנוני רגרסיית Bounding Box והקצאת מחלקות. לחלוקה, מודלים כמו U-Net או מודלים עדכניים כמו SAM מספקים תחזיות ברמת פיקסל - לא רק מה יש בתמונה, אלא גם איפה.

למשל, לקוח עשוי לרצות לא רק לזהות האם פגם קיים, אלא גם להצביע על מיקומו המדויק בתמונה. זהו לא סיווג - אלא פלט מובנה במרחב ודורש מודל מתאים.

מודלי שפה (Language Models)

דאטה טקסטואלי הוא רציף, תלוי הקשר ועשיר סמנטית. בניגוד לתמונות, הוא לא מתפרש על ידי פיקסלים אלא נשען על יחסי טוקנים, תחביר ומשמעות. הארכיטקטורה המרכזית שחוללה מהפכה בתחום ה־NLP היא ה־Transformer - מודל שמסוגל ללמוד תלות ארוכת טווח ולבצע מידול רצפים במקביל.

ה־Transformers מהווים את עמוד השדרה של מודלים לשפה כמו BERT, RoBERTa, T5, GPT ו־LLaMA. מודלים אלו מאומנים מראש על קורפוסים עצומים ואז מותאמים למשימות כמו סיווג, סיכום או דיאלוג.

בחלק מהיישומים, שליפה של מידע הופכת חיונית. במקרים כאלה משתמשים בארכיטקטורות שתומכות ב־RAG (Retrieval-Augmented Generation) - שילוב של מודל שפה עם רכיב חיפוש. בסצנות אינטראקטיביות יותר, מערכות מבוססות סוכנים מאפשרות למודל לא רק לייצר טקסט אלא גם לבצע פעולות או לשאול כלים חיצוניים.

מקרה לדוגמא: מערכת שירות לקוחות שמזהה למה התכוון הלקוח מתוך

מייל ששלח או מייצרת סיכום קצר של מסמך משפטי. המודל נדרש להבין, לתמצת ולפעול - הכל מקלט טקסטואלי בלתי מובנה.

דאטה טבלאי (Tabular Data)

דאטה טבלאי מצוי בכל תחום עסקי: יומני מכירות, יצואי CRM, רישומי פיננסים ומטא־דאטה על לקוחות. למרות שהפורמט נראה פשוט - שורות ועמודות - האתגרים המודליים אמיתיים. פיצ'רים קטגוריים, ערכים חסרים, מחלקות לא מאוזנות ומזהים בעלי קרדינליות גבוהה מסבכים את הלמידה.

מודלים מסורתיים מבוססי Ensembles כמו XGBoost ו־LightGBM עדיין שולטים בתחרויות טבלאיות רבות. הם מהירים, רובסטיים וניתנים לפרשנות. עם זאת, גישות נוירוניות מתפתחות. MLPs עובדים היטב כאשר יש כמות דאטה גדולה וארכיטקטורות כמו TabNet או FT-Transformer משלבות מנגנוני Attention כדי למודל יחסים בין עמודות בצורה יעילה יותר.

בתחומים כמו דירוג אשראי או חיזוי נטישה, פרשנות היא חובה. מגבלות רגולטוריות ותפעוליות דורשות לעיתים להצדיק מדוע התקבלה תחזית - למשל באמצעות ערכי SHAP או שיטות מבוססות Permutation.

סדרות זמן ותחזיות במידע רציף (Time Series)

דאטה טמפורלי מוסיף ציר חדש: הזמן. כל תחזית תלויה לא רק בפיצ'רים, אלא גם במה שקדם לה. חיזוי סדרות זמן כולל לרוב מגמות, עונתיות, עיכובים או פערי מדידה - ודורש מודלים שלומדים גם רצף וגם הקשר.

לשם כך משתמשים במודלים כמו LSTM ו־GRU, השומרים זיכרון פנימי לאורך זמן. מודלים מבוססי CNN כמו TCN (Temporal Convolutional Networks) מסננים על חלונות זמן ומספקים אלטרנטיבה לרקורסיה. לאחרונה וריאנטים של Transformer כמו TFT או Autoformer מציעים חלופות סקלאביליות למידול רצפים ארוכים. עבור אותות בתדירות גבוהה, כמו דאטה מחיישנים או מערכות בקרה, מודלי State Space - כגון Mamba - צוברים פופולריות.

דמיינו חיזוי ביקוש למוצר על פני מאות קודי מוצר SKU, כל אחד עם

דפוס משלו, היסטוריית מבצעים ושונות גיאוגרפית. המטרה איננה רק לזכור את העבר - אלא ללמוד כיצד הוא מתפתח ומתי הוא משתנה.

כיצד לבחור תחום להמשך

לכל תחום יש שיקולים שונים - לא רק במבנה הדאטה, אלא גם במטרות המודל, מגבלות האינפרנס ואינטגרציה מערכתית. אין פתרון אחד שמתאים לכולם.

כלל אצבע פשוט:

- אם הקלט שלכם הוא תמונה - התחילו ממודל קונבולוציה.
- אם אתם עובדים עם טקסט חופשי - בדקו Transformers.
- אם יש לכם שורות ועמודות - שקלו מודלי Ensemble או MLPs.
- אם הדאטה שלכם מתפתח לאורך זמן - חקרו ארכיטקטורות רציפות: מבוססות זמן או מבוססות Transformer.

כרגיל, מבנה הדאטה צריך להנחות את הבחירה - לא הפופולריות של האלגוריתם.

סיכום

למידה עמוקה היא משפחה של גישות המותאמות לסוגי דאטה שונים ואתגרים מודליים מגוונים. ראייה, שפה, טבלאות ורצפים - כל אחד דורש דרך חשיבה אחרת. בחירת תחום התמחות תלויה בדומיין, במטרות ובטבע הבעיה. יש מהנדסים שמעמיקים במודאליות אחת. אחרים לומדים לשלב ביניהן.

בחרו את הכלים לפי צורת הדאטה שלכם - לא לפי האופנה בתחום.

שיעור 23

מפת דרכים למהנדס DL בתעשייה

מבוא

ברכות - הגעתם לשיעור האחרון בקורס יסודות הלמידה העמוקה המודרנית. עברנו דרך ארוכה: מהעבודה הפנימית של רשת נוירונים ועד מימוש שירות תחזיות בזמן אמת בענן.

כעת הגיע הזמן לעצור ולשאול שאלה עמוקה יותר: כיצד נראית הדרך קדימה עבור מהנדס למידה עמוקה בעולם האמיתי?

שיעור זה אינו עוסק בקוד חדש או במודלים חדשים. הוא עוסק בגישה מנטלית. בבניית מערכות שמחזיקות לאורך זמן, בקבלת החלטות שניתן לעמוד מאחוריהן ובהפיכת עבודה מניסוי להשפעה ממשית.

ממודל למערכת פרודקשן

אימון מודל מדויק הוא רק ההתחלה. בסביבות תעשייתיות, דיוק בלבד אינו מספיק. מודל טוב הוא מודל שניתן לממש, לנטר, לעדכן ולסמוך עליו לאורך זמן.

משמעות הדבר היא שינוי חשיבה מ־"המודל" ל־"המערכת". קובץ עם משקולות איננו פתרון. הטמעה מלאה - מאימון וולידציה ועד שירות וניטור - הוא זה שהופך למידה עמוקה לאופרטיבית באמת.

שלושת רכיבי הליבה של מערכת DL

לכל מערכת DL בעולם האמיתי יש שלושה רכיבים יסודיים:

הראשון הוא רכיב האימון. כולל דאטה, קונפיגורציית היפר־פרמטרים, ארכיטקטורת מודל ויכולת שחזור ביצועים. תהליך אימון מתוכנן היטב מאפשר לחזור על ניסויים חודשים מאוחר יותר - עם אותן תוצאות.

השני הוא רכיב הולידציה והניטור. זה מה שמספר אם המודל עדיין עובד טוב על דאטה חדש, אם יש ירידה בביצועים ואם נדרשת פעולה. ללא ניטור מתמשך, מודל שהיה טוב עלול להפוך למסוכן בשקט.

השלישי הוא הממשק - לרוב REST API או שירות סטרימינג - דרכו צורכים את התחזיות. זה מה שמערכות אחרות רואות. עליו להיות מהיר, יציב ושקוף.

שלושת החלקים הללו חייבים להיות מתואמים ומבוקרים. יחד הם מהווים את היסוד של כל מערכת ML בפרודקשן.

מקרה לדוגמה: סיווג כוונת אימיילים

דמיינו שאתם נדרשים לבנות מודל שמזהה כוונת משתמש מתוך אימיילים נכנסים מלקוחות:

- מודול אימון שרושם לוג לכל ניסוי, כולל:
  - גרסת הדאטהסט
  - היפר־פרמטרים
  - מדדים סופיים
- מודול ולידציה ש:
  - בודק את המודל מדי שבוע על טיקטי תמיכה חדשים
  - מתריע על ירידה בדיוק או בכיסוי
- API שמחזיר:

- תחזיות
- ציוני ביטחון
- תיאור טקסטואלי קצר

כל זה מנוטר באמצעות MLflow או מערכת מטא־דאטה מבוססת Git, כך שכל שינוי מתועד וניתן להחזרה. זהו הסטנדרט ללמידה עמוקה ברמת פרודקשן.

טיפים למהנדסים בעולם האמיתי

לעבוד באופן מקצועי בלמידה עמוקה זה לא רק לכתוב קוד טוב. נדרשת משמעת. תמיד תעדו את מה שאתם עושים. לכל גרסת מודל חייב להיות תאריך ברור, קונפיגורציה ותוצאות הערכה. ודאו שחזוריות. מישהו - לעיתים אתם עצמכם - יצטרך להריץ את אותו אימון בעוד חצי שנה ולקבל אותה תוצאה. השתמשו בפורמטי סריאליזציה רובסטיים כמו TorchScript, ONNX, או SavedModel. הימנעו מסקריפטים מותאמים אישית או שמירות אקראיות.

לעולם אל תדלגו על ולידציה. מודל שמתנהג בחוסר עקביות, או שמשנה התנהגות לאורך זמן, אינו אמין. תכננו תהליכים מודולריים. ה־API שלכם לא צריך לדאוג אם המודל מתחתיו השתנה. לוגיקת האימון צריכה להיות מבודדת מלוגיקת השירות. ותמיד העריכו עקביות לאורך זמן - לא רק דיוק חד־פעמי.

אתיקה ואחריות

למידה עמוקה יכולה להשפיע על גיוס עובדים, אבחון רפואי, תמחור ועוד. עם כוח מגיעה אחריות. לעולם אל תממשו מודל שלא נבדק על פני אוכלוסיות שונות של נתונים. תמיד הבינו מאיפה הגיע הדאטה - ומה ייתכן שחסר בו. היו מוכנים להסביר תחזיות, גם אם רק חלקית. והכי חשוב: שאלו את עצמכם האם הייתם סומכים על המערכת הזו אילו היא הייתה משפיעה

עליכם או על מישהו שיקר לכם. Responsible AI איננו צ'קבוקס חוקי, אלא ערך הנדסי.

מקצוענות אמיתית

מה שמבדיל מהנדס מחובבן אינו רק עומק המודלים שלו - אלא עומק החשיבה שלו. מהנדס DL מקצועי לא רק מכוון היפר־פרמטרים. הוא בונה מערכות שהן עמידות, ניתנות לשחזור, ניתנות לניטור ובטוחות. זה דורש תשומת לב לפרטי וזה מה שהופך מודל מבטיח לפתרון אמיתי.

מניפסט למהנדס DL

העקרונות הבאים מסכמים את הערכים והגישה של מהנדס DL מקצועי בעולם האמיתי. אלו אינם חוקים - אלא הרגלים שנבדקו ומבדילים מערכות בוגרות מפרוטוטייפים שבריריים:

- **כל מודל הוא מערכת.** המודל טוב רק כמו הדאטה, תהליך הולידציה וסביבת המימוש שבה הוא חי.
- **שחזוריות מעל אינטואיציה.** כל ניסוי חייב להיות ניתן למעקב. מה שלא ניתן לשחזר - לא ניתן לסמוך עליו.
- **גרסאות.** דאטה, קוד, מודלים, קונפיגורציות, מדדים - עקבו אחרי כולם. קיצור הדרך של היום הוא הבאג של המחר.
- **ולידציה מעבר לדיוק.** העריכו גם Drift, יציבות, רובסטיות וביטחון - לא רק מדד יחיד על דאטהסט סטטי.
- **תכננו לכשל.** ניטור בלבד אינו אופציה. הניחו שדברים לא יעבדו חלק. בנו התראות, ניסיונות חוזרים ואסטרטגיות גיבוי.
- **פרשנות.** אם מודל מקבל החלטה שאינכם יכולים להסביר - לא בטוח שמותר לו לקבל החלטות בעולם האמיתי.

- **בחרו כלים לפי הדאטה - לא לפי טרנד.** תנו לצורת הבעיה להנחות את הארכיטקטורה - לא למה ש"חם" ב־arXiv.
- **Bias הוא באג.** חפשו אותו באופן אקטיבי. טפלו בו מוקדם. תעדו את דרכי ההתמודדות שלכם.
- **בנו אמון.** העבודה שלכם היא לא רק לאפטם מדדים - אלא ליצור מערכות שאחרים יכולים לסמוך עליהן.

אלו הם העקרונות שילוו אתכם כשתעברו מבניית מודלים להנדסת מערכות אמינות. שמרו אותם.

לאן ממשיכים מכאן

הקורס הזה נתן לכם את היסודות. עכשיו יש לכם את המונחים, הכלים והביטחון להתחיל פרויקטים אמיתיים. מכאן, תוכלו להעמיק: לראייה ממוחשבת, מודלי שפה, למידה טבלאית או סדרות זמן. תוכלו להתמחות - או לשלב. תוכלו לעבוד לבד - או בצוות. אך בכל דרך שתבחרו, אתם כבר מצוידים בצורת החשיבה הנכונה: שיטתית, אתית ויישומית.

סיכום

קריירה בלמידה עמוקה מתפתחת כל הזמן. אך עקרונות ההנדסה הנכונה נשארים: ללמוד, לבנות, למדוד, לשתף. בחרו פרויקט קטן ואמיתי. התייחסו אליו ברצינות. תעדו את ההנחות. ממשו את הרעיונות. אני מקווה שהקורס נתן לכם לא רק ידע - אלא גם ביטחון, כלים ומוטיבציה להמשיך. העולם זקוק למערכות שתבנו - ואתם מוכנים.

בהצלחה!