# למידה עמוקה

## הקדמה

הלמידה העמוקה (Deep Learning) היא תחום מחקר בעולם המחשבים וספציפית בתחום "למידת המכונה" שמניח שהמחשב יכול ללמוד וללמד את עצמו, ממש כמו המוח האנושי. מטרתו הברורה של התחום הזה היא ליצור חיקוי ממוחשב של פעולת המוח האנושי.

בין שנות ה-70 לאמצע שנות ה-2000, רווחה אכזבה מאד גדולה מהביצועים של תחום הבינה המלאכותית. התחום שפעם הבטיח מהפכה של ממש בעולם המחשוב, "לא סיפק את הסחורה". התקופה הזו כונתה אז "החורף של הבינה המלאכותית". אבל בשלב מסוים, בשנות האלפיים ואחרי שנים רבות של אכזבה, הפציעה האינטליגנציה המלאכותית מחדש, כשהפעם היא החלה לממש סוף כל סוף את ההבטחה מימי התום שלה. מחשבים מהירים פי מיליון, כמויות מידע עצומות שהחל להציע האינטרנט ושנאגרו על כונני ענק זולים ונגישים - כל אלה הבשילו לטכנולוגיה חזקה ומעשית - "הלמידה העמוקה".

המיוחד במערכות למידה עמוקה הוא היכולת שלהן ללמוד ולהשתפר כל הזמן ודי בעצמן. מפתחי מערכות כאלה בונים בשבילה מעין "רשת סמנטית". זו מערכת שמחקה את המערכת הנוירונים שבמוח האנושי וכאמור פועלת ולומדת כמוהו - ככל שהיא פועלת וככל שמשתמשים בה - היא משתפרת ו"יודעת" יותר.

חשיבה עמוקה היא רק בתחילתה, אבל חוקרים נעזרו בה כבר לזיהוי מולקולות שייקשרו למטרות בגוף טוב יותר ומסתייעים בממצאים לפיתוח תרופות חדשות. מחשב או תוכנה שמצוידים ב"חשיבה עמוקה" מבינים כבר היום שפה אנושית במנועי חיפוש, בחיפוש קולי ובעוזרים דיגיטליים כמו "סירי" של חברת אפל ו"קורטנה" של מיקרוסופט. בעתיד יסיקו מערכות כאלה גם מסקנות ויקבלו החלטות בכוחות עצמן. העתיד מבטיח מפיתוחים אלה מערכות רפואיות שמטפלות בחולים, אנליסטים ומומחים ממוחשבים לניהול כספים ומסחר במניות ועד למערכות ראיה ממוחשבת, שיוכלו לזהות עצמים במרחב ולסווג אותם, ללא צורך בהתערבות אנושית. באמצעות חיקוי היכולת האנושית ללמוד, מצליחות מערכות למידה עמוקה גם ליצור ציורים, מוסיקה, סרטונים וטקסטים, להמליץ ללקוחות על רכישות, לייצר קריינות מלאכותית, לזהות עצמים ופנים מדויקות בתמונות וסרטונים, לאבחן מחלות, להפיק רווחים ממניות, להלביש פנים של מפורסמים על סרטי וידאו ועוד שלל יישומים. לטווח רחוק מטרת התחום היא פיתוח של מחשב שיוכל להחליף את החשיבה האנושית. מערכת כזו תהיה מסוגלת לזהות תבניות ודפוסים בדיבור, תמונות, צלילים ועוד סוגי מידע, שהמערכות מתקשות כיום לעבד ולטפל בהם.

בשנים האחרונות נבנו חומרות מיוחדות (GPU) שמטרתן להריץ אלגוריתמים בלמידה עמוקה מהר יותר ממעבד רגיל.

## היסטוריה

1950: רשתות עצביות (perceptron) הומצא על ידי רוזנבלט.

1980/90: רשתות עצביות נהיות פופולאריות ואז ננטשות כרעיון מעניין אך בלתי אפשרי או "חסר עקרון".

1990: LeCun משיג ביצועים חדישים על זיהוי תווים באמצעות רשת קונבולוציה (רעיון מרכזי ברשתות של בימינו).

2000: Hinton, Bottou, Bengio, LeCun, Ng, ואחרים ממשיכים לנסות דברים עם רשתות עמוקות אך ללא הרבה הצלחה.

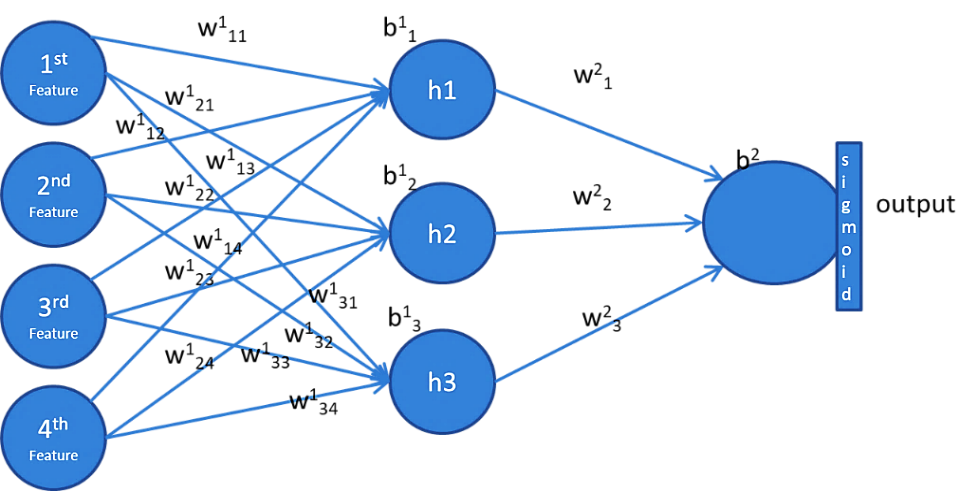
2010-2011: התקדמות משמעותית באזורים מסוימים, אך קהילת הראייה עדיין לא משוכנעת. חלק מהחוקרים כועסים על התעלמות / דחייה

2012: הלם מביצועי רשת נוירונים בכנס ECCV 2012 עם אתגר ImageNet.

# Multi Layered Perceptron

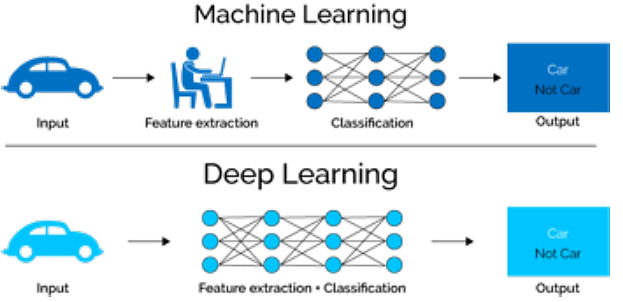
## הגדרה

זוהי טכניקה שבאמצעותה יוצרים מהתכונות הראשוניות שלנו תכונות חדשות הנקראות נוירונים (או perceptron’s), ובכך ליצור שכבה נוספת של תכונות. מכל שכבת נוירונים ניתן לבנות שכבה נוספת של נוירונים. כל שכבת נוירונים נקראת "שכבה חבויה" (Hidden Layer). בשכבה האחרונה נשתמש ברגרסיה ליניארית, לוגיסטית או Softmax כדי לקבל את הניבוי של המודל. הרעיון לבנות מודל הבנוי על שכבות נוירונים מגיע מהצורה שבה עובד המוח האנושי והדרך בה הוא משדר מידע.

נוכל להגדיר כמה שכבות שנרצה וכן להגדיר כמה נוירונים שנרצה בכל שכבה. ככל שיש יותר שכבות כך אנו יכולים ליצור פונקציות מורכבות ויעילות יותר, אולם אז אנו מסתכנים ב-overfitting. מספר השכבות ומספר הנוירונים בכל שכבה הם היפר-פרמטר, כלומר יש לבדוק מספר ארכיטקטורות של שכבות, עם צירוף של רגולריזציה, ולבחור את הטובה ביותר. כל נוירון מייצג איזשהו צירוף ליניארי של **כל** הנוירונים בשכבה הקודמת ונוצר מתהליך למידה של המודל. כל שכבה היא fully connected לשכבה הקודמה לה. מטריצת המשקלים בשכבה ה-i היא בגודל , כאשר הוא מספר הנוירונים בשכבה ה-i ו- היא מספר הנוירונים בשכבה ה-i-1. השכבה ה- מתקבלת מהחישוב .

שיטה זו מאוד יעילה, שכן פעמים רבות התכונות הראשוניות לא מספיקות וצריך ליצור תכונות חדשות, משמעותיות יותר למה שאנו רוצים לנבא. אולם כיוון שהתכונות שנרצה ליצור משתנות ממקרה למקרה שיטת רשת נוירונים עושה זאת בשבילנו בצורה טובה. למידה עמוקה מספקת מסגרת מאוד גמישה ואוניברסלית, ללמידת העולם. לדוגמה, מודל המקבל שני מספרים ומחזיר האם המספרים רחוקים או קרובים. במודל רגיל שבו נותנים משקלים קבועים לכל תכונה תהיה בעיה, שכן יכול להיות שנקבל x גדול ו-y קטן פעם כך (x, y) ופעם כך (y, x). לעומת זאת, רשת נוירונים תדע ליצור נוירון המייצג את תכונות ההפרש בערך מוחלט או בריבוע. יצירת תכונות משמעותיות יותר מאשר התכונות הראשוניות יוצרת בעיה חדשה עם מימד נמוך יותר שבה יותר קל לסווג ולהפריד בין האובייקטים בדאטה.

## הבדלים בין למידת מכונה ללמידה עמוקה

בלמידת מכונה מסתמכים על המתכנת שיכניס תכונות לאובייקטים ויעצב את האלגוריתם המתאים. לדוגמה, בבעיית זיהוי של תפוחים בתמונות המתכנת יצטרך להתחשב בצבע, גודל, צורה, מרקם וכו', המודל צריך ללמוד מתכונות אלו בלבד. בלמידה עמוקה, לעומת זאת, נותנים למודל אפשרות להוסיף תכונות משלו על ידי צירוף כלשהו של מספר תכונות.

## Activation function

למידה עמוקה היא בעצם שרשור מודלים ליניאריים, ולכן כל נוירון הוא צירוף לינארי של כל הנוירונים בשכבה הקודמת. לדוגמה, עבור הנוירון הראשון בשכבה 1 נקבל:

.

אם נשאיר זאת כך נקבל שהיינו יכולים לבנות את כל המודל באמצעות שכבה אחת ליניארית, כמו במודלים הפשוטים שאינם למידה עמוקה. לכן, לאחר שחישבנו שכבת נוירונים באמצעות השכבה הקודמת נפעיל עליה פונקציית אקטיבציה, שתשנה מעט חלק מהמשקלים, כדי לשבור את הליניאריות ותאפשר ללמוד מגוון רחב של פונקציות שהן לא ליניאריות. ישנם מספר פונקציות אקטיבציה, בדרך כלל משתמשים ב-ReLU או LeakyReLU.

1. ReLU - מאפסת כל משקלים הקטנים או שווים ל-0. מפשטת את חישוב הגרדיאנט ועוזרת למנוע בעיית vanishing gradient, שבה הערכים הולכים וקטנים עד שנעלמים. מתכנס מאוד מהר וקל לחישוב. הנגזרת החלקית לכל נוירון יהיה 1 אם ערכו חיובי ו-0 אם שלילי. בעייתי כאשר הנוירון תמיד יוצא שלילי ואז אי אפשר לעדכן את המשקלים עליו (נוירון מת). כדאי בדרך כלל להתחיל עם פונקציה זו.
2. LeakyReLU - מכפיל את כל המשקלים הקטנים או שווים ל-0 באיזשהו קבוע קטן . *כאן הנגזרת לא מתאפסת עבור ערך שלילי ואפשר לעדכן אותה. בדרך כלל יציג יעילות דומה ולעיתים רחוקות קצת יותר טובה מ-*ReLU.
3. ELU - שילוב בין שתי הפונקציות הקודמות. לרוב ELU לעיתים רחוקות יותר יעיל אבל בגדול אין הרבה הבדל.
4. Sigmoid - כמו פונקציית הסיגמואיד מ-logistic regression. מחזירה ערך בין 0 ל-1. החיסרון שעבור ערכים גדולים מ-5 וקטנים מ-5- יחזיר ערכים ששואפים ל-1 ו-0 שאינם מאפשרים לקבל מגוון רחב של ערכי הגרדיאנט וכך תהליך הלמידה פחות טוב. בנוסף, כאשר הערכים תמיד חיוביים אזי הגרדיאנט הוא או חיובי או שלילי. במצב כזה התנועה לכיוון הגרדיאנט תהיה תמיד בצורת זיג-זג ולא באופן ישיר, כלומר תהליך הלמידה לא יעיל מסיבות אלו אף פעם לא נרצה להשתמש בsigmoid כפונקציית אקטיבציה.
5. Tanh - מחזירה ערך בין 1- ל-1. גם כאן אותה בעיה כמו ב-sigmoid, שכמעט תמיד מחזיר ערכים ששואפים ל-1 ו-1- שאינם מאפשרים לקבל מגוון רחב של הגרדיאנט.

אם, כן כדי לחשב את החיזוי הסופי של רשת נוירונים עם n שכבות, נצטרך לבצע את החישוב הבא.

במקרה של סיווג בינארי יש להכניס ערך זה לפונקציית Sigmoid, ובמקרה של סיווג ליותר משני מחלקות מכניסים כל הערכים ל-Softmax. תהליך זה של חישוב מהשכבה הראשונה עד האחרונה כדי לקבל את החיזוי הסופי נקרא forward pass.

## Backpropagation

כדי לאמן את הרשת אנו צריכים בכל איטרציה לעדכן את המשקלים בהתאם לכיוון הנגדי לגרדיאנט. לשם כך אנו צריכים לחשב את הגרדיאנט של כל משקל ברשת w בפונקציית ה-loss, כלומר לחשב את , ואז לעדכן . האלגוריתם איתו מחשבים את הגרדיאנט לכל משקל ברשת נוירונים נקרא Backpropagation, המסתמך על כלל השרשרת בחישוב נגזרת של פונקציה מורכבת. לפי כלל השרשרת הנגזרת של פונקציה מורכבת היא .

מספר הגדרות שיעזרו לנו בהסבר האלגוריתם:

– זה המשקל בשכבה l מצומת k (הנמצאת בשכבה l-1) אל הצומת j.

*– ה-*bias *של צומת* j *בשכבה* l*.*

*– הפלט של צומת* j *בשכבה* l*.*

*– הפלט של צומת* j *בשכבה* l *לאחר הפעלת פונקציית האקטיבציה על* . *נשים לב כי מתקיים:*

– הנגזרת של פונקציית ה-loss לפי , כלומר . ערך זה נקרא local gradient או delta error.

נוכיח שאם יש לנו את נוכל לחשב את כל המשקלים וה-bias של צומת j בשכבה l. נשתמש בכלל השרשרת:

נותר אם כן להסביר כיצד יש לחשב את לכל שכבה l ולכל צומת j. נחלק לשני מקרים:

1. אם l היא השכבה האחרונה אזי:

שני נגזרות אלו קל לחשב שכן הנגזרת השמאלית היא הנגזרת של פונקציית ה-loss (כמו Softmax, Sigmoid, tanh), והנגזרת הימנית היא הנגזרת של פונקציית האקטיבציה.

1. אם l היא אינה השכבה האחרונה אזי:

*מכאן, שאם יש לנו את כל לכל צומת בשכבה* l+1 *נוכל לחשב את כל הגרדיאנטים בשכבה* l*. ככה נעבור בהתפשטות מהשכבה האחרונה עד לראשונה, נחשב את כל הגרדיאנטים ונעדכן את המשקלים.*

האלגוריתם הסופי לאימון רשת נוירונים הוא:

Repeat until convergence:

For every example in data: (or better mini batch)

* Perform forward pass. Get all z, a.
* Compute Error (final loss)
* Compute all δl
* Compute for all w and b: and .
* Preform update. For all w, b:

## אתחול משקלים

אנו מעוניינים שהמשקלים בכל שכבה לא יהיו קבועים אלא רנדומליים. שכן ערכים קבועים יביאו לכך שכל הנגזרות יהיו זהות ואז בכל שכבה כל הנוירונים יהיו זהים. בנוסף, נרצה שממוצע המשקלים יהיה סביב ה-0 ולא שכל המשקלים יהיו אותו סימן (חיובי או שלילי), שכן אם כל המשקלים יהיו מאותו סימן אזי הגרדיאנט הוא או חיובי או שלילי. במצב כזה התנועה לכיוון הגרדיאנט תהיה תמיד בצורת זיג-זג ולא באופן ישיר, כלומר תהליך הלמידה לא יעיל. כעת נותר לקבוע האם אנו רוצים משקלים קטנים או גדולים.

אם המשקלים יהיו קטנים מדי אזי ברשתות עמוקות הערכים ילכו ויתקרבו ל-0. במצב כזה יש חשש שהגרדיאנט יהיה מאוד קטן וכתוצאה מכך תהליך הלמידה יהיה מאוד איטי או ייעצר (Vanishing Gradient). לעומת זאת, אם המשקלים יהיו גדולים מדי, אזי בפונקציות אקטיבציה של sigmoid או tanh, נקבל בכל הנוירונים אותם ערכים ואז הגרדיאנט יהיה מאוד קטן ושוב תהליך הלמידה יהיה איטי מאוד או ייעצר.

הפתרון הוא למצוא משקלים שהם לא קטנים מדי ולא גדולים מדי. שיטה טובה שעושה זאת היא Xavier Initialization, שמאתחלת משקלים בכל שכבה לפי מספר הנוירונים באותה שכבה. אם פונקציית האקטיבציה היא ReLU אז מומלץ להשתמש בשיטת MSRA Initialization שדומה מאוד ל--Xavier Initialization.

### נוירון 0

נרצה שה-bias בכל שכבת נוירונים יהיה חיובי, שכן אם בנוירון כל הערכים ייצאו שליליים וגם נשתמש ב-ReLU, ערך הנוירון ייצא 0 ואז נקבל נוירון מת שלא יכול לשנות משקולות. לכן תמיד נאתחל את ה-bias לערך חיובי, בדר"כ באזור 0.1.

## דוגמת קוד ב-TensorFlow

מודל המקבל תמונה של ספרה הכתובה בכתב יד ומחזיר איזו ספרה זו. הדאטה עליו המודל מתאמן הוא MNIST המכיל 60,000 תמונות כאלו. כל תמונה יש פיקסלים. כל ספרה מיוצגת ב-hot encoded vector, כלומר וקטור בגודל 10 שבו יש אחד במיקום של הספרה המתאימה. המודל בנוי על שני שכבות נוירונים בגודל 100 ו-50. בסוף יש שכבת Softmax המחזירה הסתברות שייכות לכל ספרה. לאחר מיליון הרצות המודל מדייק ב-97% מהמקרים.

(hidden1\_size, hidden2\_size) = (100, 50)

x = tf.placeholder(tf.float32, [None, 784])

y\_ = tf.placeholder(tf.float32, [None, 10])

W1 = tf.Variable(tf.truncated\_normal([784, hidden1\_size], stddev=0.1)) // first layer

b1 = tf.Variable(tf.constant(0.1, shape=[hidden1\_size]))

z1 = tf.nn.relu(tf.matmul(x,W1)+b1)

W2 = tf.Variable(tf.truncated\_normal([hidden1\_size, hidden2\_size], stddev=0.1)) // second layer

b2 = tf.Variable(tf.constant(0.1, shape=[hidden2\_size]))

z2 = tf.nn.relu(tf.matmul(z1,W2)+b2)

W3 = tf.Variable(tf.truncated\_normal([hidden2\_size, 10], stddev=0.1)) // Softmax

b3 = tf.Variable(tf.constant(0.1, shape=[10]))

y = tf.nn.softmax(tf.matmul(z2, W3) + b3)

cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(y), reduction\_indices=[1]))

train\_step = tf.train.GradientDescentOptimizer(0.001).minimize(cross\_entropy)

init = tf.global\_variables\_initializer()

sess = tf.Session()

sess.run(init)

correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

for i in range (1000):

for \_ in range(1000):

batch\_xs, batch\_ys = mnist.train.next\_batch(100)

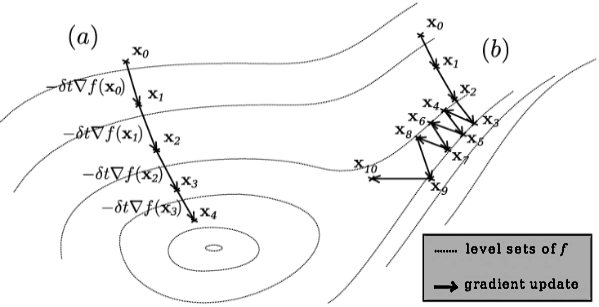
sess.run(train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys})

print(i, sess.run(accuracy, feed\_dict={x: mnist.test.images, y\_: mnist.test.labels}))

# Optimization

## Gradient Descent

Gradient descent (בתרגום מילולי: מורד הגרדיאנט) היא שיטת אופטימיזציה איטרטיבית מסדר ראשון למציאת מינימום מקומי של פונקציה. וקטור הגרדיאנט זהו וקטור שמייצג את הנגזרת של פונקציה עם מספר משתנים, ומסומן ב-. בכל כניסה בוקטור נמצא הנגזרת של הפונקציה לפי משתנה אחר. גודל הוקטור כמספר המשתנים בפונקציה. אם נציב בוקטור הגרדיאנט נקודה כלשהי, אזי הכיוון של וקטור הגרדיאנט הוא השיפוע המקסימלי באותה נקודה, כלומר הכיוון בו נמצא השינוי הדרסטי ביותר בין הנתונים סביב נקודה מסוימת. לעומת זאת, הכיוון ההופכי לגרדיאנט הוא הכיוון בו השיפוע הוא הנמוך ביותר.

הרעיון בשיטת ה- Gradient Descentהוא להתחיל בערכים שרירותיים כלשהם של המשקלים, ובלולאה לקדם את כל המשקלים בכיוון ההפוך לגרדיאנט של פונקציית ה-loss בצעדים של קצב למידה , עד שמגיעים למצב של "התכנסות" (convergence) או נקודה מספיק נמוכה המוגדרת בתנאי העצירה. "התכנסות" זה מצב שבו הצבתם של המשקלים הנוכחיים בוקטור הגרדיאנט של פונקציית ה-loss ייתן לנו את וקטור ה-0 או שואף לכך. כאשר מגיעים למצב של התכנסות מצאנו את המשקלים המינימליים. דבר זה דומה לאדם העומד בנקודה על המפה הטופוגרפית אך ישנו ערפל סמיך אשר עוצר בעדו. לכן באפשרותו לבדוק רק בסביבה הקרובה לו היכן הזווית הכי תלולה של המדרון ודרכה הוא יורד.

חשוב שקצב הלמידה לא יהיה גדול מדי, שכן אז עלולים לפספס את המינימום ובצעד חזור שוב לפספס, וכך אף פעם לא להגיע למינימום, לכן צריך ש- יהיה קטן יחסית. קצב הלמידה הוא היפר-פרמטר, אמנם בדר"כ יהיה באזור 0.001.

### קוד פייתון המממש Gradient Descent

*# x0 - initial guess*

*# df - gradient of function*

**def** gradient\_descent(x0, df)

cur\_x = x0

alpha = 0.01  *# learning rate*

precision = 0.00001

previous\_step\_size = cur\_x

**while** previous\_step\_size > precision:

prev\_x = cur\_x

cur\_x += -alpha \* df(prev\_x)

previous\_step\_size = abs(cur\_x - prev\_x)

**return** cur\_x

### תיאור מתמטי

נתונה פונקציית loss של מודל , כאשר הם המשקלים של המודל. את קבוצת כל המשקלים באיטרציה ה-i של Gradient Descent נסמן . וקטור הגרדיאנט של הוא . העדכון של כל המשקלים בכל איטרציה i ב-Gradient Descent יהיה לפי הנוסחה:

*נשים לב שפונקציית ה-*loss *משתנה כל איטרציה. עבור קצב למידה קטן דיו, נקבל שמתקיים . במילים אחרות, הביטוי מוחסר מ- כיוון שרוצים לזוז נגד כיוון הגרדיאנט, מטה לכיוון המינימום. בהתבסס על הבחנה זו, ניתן לנחש משקלים ראשוניים , ולקבל את הסדרה , שלפי ההבחנה לעיל מקיימת את אי-השוויון: , כלומר פונקציית ה-*loss *תלך ותקטן עד שנגיע להתכנסות או טעות קטנה של המודל.*

### Analytical and Numerical Gradient

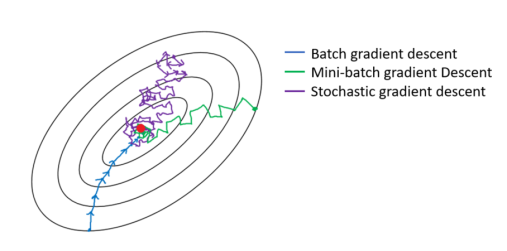
אלו שיטות לחשב את וקטור הגרדיאנט של פונקציה f(x). ב- Numerical Gradientמחשבים כל ערך בוקטור הגרדיאנט לפי נוסחת הגבול לחישוב נגזרת:

כלומר, עבור כל משקל , מוסיפים ל- ערך קטן h, מחשבים את ה-loss מחדש, מחסרים ממנו את ה-loss המקורי ומחלקים ב-h. שיטה זו מאוד איטית (שכן צריך לחשב את ה-loss עבור כל משקל) וגם פחות מדויקת אך קלה למימוש.

ב- Analytical Gradientלעומת זאת, מחשבים את הגרדיאנט בצורה יותר מהירה ומדויקת אך יותר קשה למימוש. תמיד נרצה לממש את חישוב הגרדיאנט לפי Analytical Gradient, אך נבדוק את המימוש באמצעות השוואה ל-Numerical Gradient. השוואה זו נקראת “Gradient Check”.

## חלוקת ה-Train

יש מספר דרכים כיצד להריץ אלגוריתם למידה על ה-train:

* Batch Gradient Descent (BGD) - משתמש בכל הדאטה כדי לחשב את ה-loss ועליו לחשב את הגרדיאנט. במידה והמידע כולו נכנס בזיכרון נעדיף בדרך כלל להשתמש בשיטה זו, שכן הצעדים בה הם יותר מדויקים.
* Stochastic Gradient Descent (SGD) - מערבב את המידע בתחילת כל איטרציה, לאחר מכן לוקח חלק קטן מהמידע באופן אקראי, מחשב רק עליו את ה-loss ומבצע צעד בכיוון הגרדיאנט. היתרון של שיטה זו הוא שחישוב הגרדיאנט הרבה יותר מהיר מאחר ומשתמשים בחלק קטן מהדאטה. החיסרון שדורש הרבה איטרציות מאחר וכל איטרציה פחות מדויקת, אולם הוכח שעבור מספיק איטרציות משיג תוצאות יעילות כמו BGD.
* Mini-Batch Gradient Descent (MB-GD) - מערבב את המידע פעם אחת בהתחלה וכל איטרציה לוקח חלק אחר שלא השתמש בו עד כה. על חלק זה מחשב את ה-loss וממנו את הגרדיאנט. משתמשים בשיטה זו כאשר המידע מאוד גדול.

## Batch-Normalization

בשיטת SGD ו-MB-DS בכל איטרציה יש מידע שונה שלפיו מעדכנים את המשקולות. לכן הקלט לכל נוירון יכול להיות שונה בכל איטרציה. בשכבות הראשונות לשינויים אלו אין השפעה, אך בשכבות עמוקות זה יוצר מצב שהתפלגות הקלטים לכל נוירון יכולים להיות שונים מאוד. כתוצאה מכך, נוירונים אלה צריכים להסתגל באופן רציף להתפלגות הקלט המשתנה. זה יכול לגרום לאלגוריתם הלמידה לרדוף אחר מטרה נעה ולא ללמוד מספיק טוב.

בעיה נוספת, בשכבות העמוקות הפלט של הנוירונים יכול להיות מאוד גבוה או מאוד נמוך, ואז אם משתמשים בפונקציות אקטיבציה Sigmoid או tanh, הפלט הסופי יהיה זהה מה שגורם לגרדיאנט להיות 0 ואין תהליך למידה.

Batch-Normalization פותר בעיות אלו בכך שמנרמל את שכבת הנוירונים. עבור כל שכבה שמפעילים עליה Batch Norm צריך לחשב ממוצע ושונות של כל הנוירונים. לכל נוירון מחסרים ממנו את הממוצע ומחלקים בשונות. הנרמול מבטיח שכעת הממוצע לכל הנוירונים הוא 0 והשונות הוא 1, כך שהקלט לכל נוירון בשכבה הבאה תמיד יהיה מאותו סדר גודל, בלי קשר ל-batch. אולם בעיה חדשה שאם כל שכבה מנורמלת אזי שינויי המשקלים שנעשו על ידי השכבה הקודמת והרעש בין ה-batches יכולים להיעלם כתוצאה מהנרמול. זה יכול להוביל להעברת משקולות לא אופטימליות. כדי לתקן זאת מוסיפים שני פרמטרים ניתנים לאימון, גמא ובטא , לפלט של כל נוירון משכבה קודמת שיכולים לשנות את הערך המנורמל ואף להחזיר אותו למצבו הקודם אם שווה לשונות ו שווה לממוצע.

כל מה שתיארנו לעיל נעשה בשלב ה-train, בשלב ה-test לעומת זאת חישוב הממוצע והשונות נעשה על ידי כל שפגשנו בשלב ה-train באמצעות ממוצע ושונות רצים.

היתרונות הגדולים של Batch-Normalization שמאפשר להגדיל את הקצב למידה ובכך לאמן את המודל מהר יותר. בנוסף, מונע overfitting של המודל.

## Optimizers

עד כה ראינו את הגרסה הפשוטה של Gradient Descent אמנם יש עוד סוגים של אופטימייזרים. כולם מבצעים צעד בכיוון הגרדיאנט אך עם כל מיני חישובים שונים. ההבדל ביניהם הוא בדרך כלל לא באם התכנס או לא אלא במהירות ההתכנסות.

### SGD+Momentum

לשיטת הקודמות שמסתמכות רק על כיוון הגרדיאנט יש לא מעט בעיות:

1. אם יש כיוון בו פונקציית ה-loss משתנה מהר אך הגרדיאנט משתנה לאט, נוצר מצב שהצעדים נעשים בזיגזגים וההתכנסות נהיית איטית.
2. כאשר מגיעים למינימום מקומי, או נקודת אוכף הגרדיאנט שווה ל-0 ההתקדמות תיעצר. בממד גבוה זו סיטואציה שכיחה. בעיה זו נוצרת גם בסביבת נקודת האוכף שבה הגרדיאנט מאוד קטן.
3. ב-SGD או MB-GD כיוון הגרדיאנט עלול לסבול מרעשים בעקבות העבודה עם חלק מהדאטה. לדגימות חריגות (outliers) יכולה להיות השפעה חזקה.

כדי להתגבר על בעיות אלו מוסיפים ל-SGD מומנטום. תנועות של גופים בעולם הפיזי לרוב חלקות ולא נוטות לזגזג מפני שהן צוברות מהירות מהצעדים הקודמים שלהן. דמיינו כדור בשטח הררי מנסה להגיע לנקודה הנמוכה ביותר. כאשר השיפוע גבוה, הכדור תופס תאוצה ומסוגל לעבור בדרכו בגבעות קטנות בלי להיתקע. כאשר השיפוע קטן המומנטום ומהירות הכדור פוחתת, ובסופו של דבר מגיע לנוח במצב הנמוך. באותו אופן, הוספת המומנטום ל-SGD אומרת שלא מעדכנים משקלים רק על סמך הגרדיאנט הנוכחי אלא לפי מהירות v, שהיא ממוצע משוכלל של היסטוריית הגרדיאנטים. המהירות הנוכחית תלוי במהירות משלב קודם, חיכוך , קצב למידה וגרדינאט נוכחי. הוא בדרך כלל 0.9 או 0.99, תלוי כמה משקל אנו רוצים לתת למהירות קודמת.

SGD עם מומנטום מגיב פחות טוב לשינויים מהירים אך באופן כללי מספק דחיפה גדולה למהירות הלמידה וכמעט תמיד עובד מהר יותר מאשר SGD. הוספת המומנטום תחליק את צעדי הגרדיאנט ותפתור את כל שלושת הבעיות שציינו לעיל.

שיטה נוספת של מומנטום נקראת "Nesterov Momentum", שלפיה יש לחשב את הגרדיאנט הנוכחי לא מהמיקום הנוכחי אלא מהמיקום בו היינו לולא היינו לוקחים צעד לכיוון המהירות הנוכחית.

שיטה זו בחלק קטן מהמקרים יכולה לשפר משמעותית את תהליך הלמידה, אך לרוב נעדיף ללכת על בטוח ולהשתמש במומנטום הרגיל.

### AdaGrad

עד כה השתמשנו בקצב למידה לא משתנה ושווה לכל המשקלים שלנו. כעת נלמד שיטות שבהם נוכל לכוון באופן אדפטיבי את קצב הלמידה לאורך שלבי האימון ולדעת איזה משקל להאיץ ובאיזה להאט. אופטימייזרים אלו נקראים אדפטיביים.

AdaGrad שומר סכום רץ של כל ריבועי הגרדיאנטים שאנחנו רואים במהלך האימון. כאשר נרצה לעדכן את המשקל נחלק בסכום ריבועי הגרדיאנטים. סוכמים גרדיאנטים בריבוע כדי לא להתחשב בכיוון כל גרדיאנט אלא רק בגודל.

הרעיון הוא שמקטינים את קצב הלמידה עם הזמן לכל המשקלים (סכום הריבועים ממשיך לגדול), אלא שמשקלים שמקבלים עדכונים גדולים יקטינו את קצב הלמידה שלהם מהר מאוד, בעוד שמשקלים שמקבלים עדכונים קטנים יקטינו את קצב הלמידה לאט. כך אנו יוצרים אדפטציה, כלומר שוויון בין כיוונים שונים, ומגדילים צעדים לכיוונים שפחות הלכנו בהם עד כה. במקרה שהפונקציה קמורה (convex), ולכן מובטח בה מינימום יחיד, פעולה זו טובה כי אנו רוצים לעשות צעדים קטנים יותר כדי לא להחמיץ את המינימום.

החיסרון בשיטה זאת הוא כאשר סכום הריבועים גדל מדי ואז קצב הלמידה שואף ל-0 ואין התקדמות. בנוסף, במקרה שהפונקציה לא קמורה אנו עלולים להיתקע בנקודת אוכף. בפועל, AdaGrad לא כל כך שימושי בגלל הנטייה שלו להיתקע.

### RMSProp

כמו AdaGrad אלא שמטרתו לפתור את בעיית השאיפה ל-0 של קצב הלמידה בכך שמוסיף פרמטר, "קצב דעיכה", כדי להקטין את הסכום הריבועי של הגרדיאנטים ולתת יותר משקל לגרדיאנט הנוכחי. פרמטר זה הוא בדרך כלל 0.9 או 0.99.

### Adam

משלב את היתרונות של SGD+Momentum ושל RMSProp.

המהירות מיוצגת ב-first\_moment ודעיכת קצב הלמידה ב-second\_moment. הבעיה בשיטה זו היא שבאיטרציות הראשונות first\_moment ו-seccond\_moment יכולים להיות מאוד קטנים אם מאותחלים ל-0. לכן בגרסה מעודכנת של Adam מחלקים אותם בערך התלוי במספר הצעד t כך שאם אלו הצעדים הראשונים נגדיל את המומנטים.

Adam הוא האופטימייזר הטוב ביותר על מגוון רחב של בעיות, ולכן בדרך כלל מומלץ להתחיל איתו.

### RAdam

האימון הראשוני של Adam לא יציב, בגלל שבתחילת האימון יש מעט נקודות לחישוב הממוצע עבור המומנט השני. לכן צעדים גדולים המבוססים על מידע מועט ורועש יכולים לגרום להשתקעות בנקודת מינימום מקומית או אוכף.

RAdam (Rectified Adam)-) הוא אופטימייזר שמחליף את ה-warm-up ב-Adam ולא מצריך הגדרת פרמטרים עבור תקופת החימום. חוקרים שפתחו את RAdam מצאו כי בעיית ההתייצבות הראשונית נגרמת כתוצאה משונות לקויה בקצב הלמידה האדפטיבי הנוצר מכמות הנתונים המוגבלת בתחילת האימון. הדרך בה RAdam פותר את הבעיה היא על ידי תיקון (Rectified) השונות של הקצב הלמידה האדפטיבי על סמך הגרדיאנטים.

### Ranger

חידוש נוסף שיצא לאחרונה בתחום האופטימייזרים הוא LookAhead. זהו אלגוריתם שמתלבש מעל אופטימייזרים קיימים. הוא משפר את יציבות הלמידה, מוריד את השונות של האופטימייזר עליו הוא מתלבש, מתכנס מהר יותר ומפחית את הצורך בכוונון מוקפד של היפר-פרמטרים. LookAhead שומר על שתי קבוצות משקולות ומשלב ביניהם (מבצע אינטרפולציה), משקולות מהירות שמסתכלות קדימה וחוקרות את השטח ומשקולות האיטיות נשארות מאחור ומספקות יציבות. לאחר מספר צעדים חקרניים המשקולות האיטיות מתקדמות בכיוון המשקולות המהירות בצעד קטן, והמשקולות המהירות מתחילות להתקדם מהמקום החדש של המשקולות האיטיות. הצעדים החקרנים (המהירים) נקבעים על ידי האופטימייזר הפנימי עליו LookAhead מתלבש.

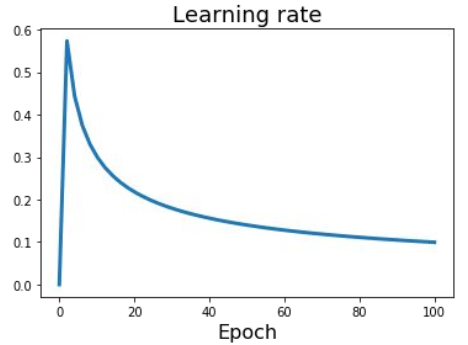
Ranger משלב את RAdam עם LookAhead. כל אחד מהם מטפל בהיבטים שונים של אופטימיזציה. RAdam מייצב אימונים ההתחלתיים, ו-LookAhead מייצב את האימונים הבאים, מבצע אקספלורציה ומזרז את תהליך ההתכנסות.

## קצב למידה

בחירת קצב למידה גבוהה מידי יכול לגרום להתבדרות או התכנסות למינימום מקומי, ואילו לקצב למידה נמוך מידי ייקח זמן רב להתכנס. לא חייבים לדבוק בקצב למידה קבוע אלא מומלץ לרוב להקטין את קצב הלמידה ככל שמתקרבים למינימום ובכך לשלב בין היתרונות של קצב לימוד מהיר ואיטי. הקטנת קצב הלמידה מקובלת לרוב בשימוש ב-SGD ופחות באופטימייזרים אדפטיביים שבהם שינוי הקצב למידה כבר מובנה בהם.

שינוי קצב הלמידה מתבצע לפי מספר ה-epoch (מספר הפעמים שעברנו על הדאטה). נסמן קצב הלמידה ההתחלתי, קצב הלמידה ב-epoch t, ו-T מספר הכולל של epochs בתהליך הלמידה. יש כמה שיטות כיצד לעדכן קצב למידה:

1. Linear:
2. Cosine:
3. Inverse Sqrt:



### Linear Warmup

בתחילת האימון יש מעט נקודות ולכן צעדים גדולים המבוססים על מידע מועט ורועש יכולים לגרום להשתקעות בנקודת מינימום מקומית או אוכף. זוהי בעיה המשותפת לרוב האופטימייזרים, במיוחד אלו שמשתמשים במומנטום. דרך אפשרית להתמודד עם בעיה זו היא להתחיל לאמן בקצב נמוך, וכאשר המודל מתגבר על בעיית ההתייצבות הראשוניות, להגביר את הקצב. חסרונה של השיטה הוא שמידת החימום הנדרש אינה ידועה ומשתנה מבעיה לבעיה.

## Early Stopping

הרעיון הוא שמפסיקים את האימון באמצע לפני שמגיעים ל-overfitting, כלומר הדיוק על ה-validation פוחת ואילו הדיוק על ה-train עולה. לשם כך האלגוריתם צריך לשמור את התוצאה הטובה ביותר שמצא, וכאשר רואה שלמשך מספר צעדים מוגדר הדיוק על ה-validation יורד, עוצרים את תהליך הלמידה ומחזירים את המודל הטוב ביותר שנמצא עד כה. ניתן לבצע early stopping באופן מובנה כמעט בכל הספריות של למידה עמוקה.

## Dropout

בשיטה זו בכל איטרציה שבו המודל **לומד** מאפסים אחוז מסוים p מהנוירונים בכל שכבה. אם לדוגמה בשכבה יש 10 נוירונים ו-p=30% אזי הפלט של שלושה נוירונים אקראיים לשכבה הבאה יתאפס באותה איטרציה. באיטרציה הבאה יתאפסו נוירונים אחרים. נשים לב שהשימוש ב-dropout הוא בשלב ה-train בלבד ולא בשלב ה-validation וה-test. כדי לשמור על הסדרי גודל בין ה-train ל-test, בשלב ה-test נכפיל תוצאת כל נוירון ב-p, אפשרות נוספת היא שבשלב ה-train נכפיל את הפלט של הנוירונים שלא יתאפסו ב- ואז אין צורך לבצע שום שינוי בשלב ה-test.

היתרונות של שיטה זו הם:

1. גורם לכל נוירון להיות משמעותי יותר שכן לא מסתמכים רק על אותם נוירונים אלא כל צעד מחזקים נוירונים אחרים.
2. מונע Overfitting שכן בכל צעד מתקבל מידע שונה מכל שכבה ואין התאמת יתר ל-train. מכאן ש-dropout יכול לשמש כ-regularization.

בפועל Dropout נותן תוצאות טובות ומומלץ להשתמש בו ברשתות עמוקות. האחוז p נקבע באופן של ניסוי וטעיה אך בדרך כלל לא יעבור את ה-50%.

ב-Tensorflow ניתן להשתמש בשיטה tf.nn.dropout המקבלת את הפלט של נוירון בשכבה כלשהי ואת האחוז p. את ההסתברות p נכניס בתור placeholder כי אנו רוצים ש-p יהיה שונה מ-0 רק בשלב ה-train ולא על מידע אחר.

prob = tf.placeholder(tf.float32)

h \_drop = tf.nn.dropout(h, prob)

## Data Augmentation

סוג נוסף של רגולריזציה הנפוץ בעיקר בעיבוד תמונה. לפי שיטה זו יש להגדיל את ה-train על ידי הוספת שינויים לדאטה הקיים והוספתו ל-train המקורי. המטרה כאן היא לאפשר למודל להתאמן על מגוון של סוגי תמונות. יעיל בעיקר כאשר יש מעט דאטה או שאינו מהווה מדגם מייצג. סוגים נפוצים של שינויים הם:

* הפוך אופקי של התמונה.
* חיתוך התמונה במקומות מסוימים.
* שינוי הצבעים: ניגודיות, בהירות, הוספת קבוע לכל הפיקסלים.
* שינוי התמונה: חיתוך, סיבוב, עמעום, שקיפות.
* איפוס הפיקסלים באזור רנדומלי כלשהו בתמונה ל-0.

## קביעת היפר-פרמטרים

יש לבצע מספר צעדים כדי למצוא את ההיפר-פרמטרים הטובים ביותר:

1. בשלב ראשון נרצה בכלל לבדוק שאפשר ללמוד על הדאטה שלנו. לשם כך ניקח חלק מאוד קטן מהדאטה וננסה להגיע ל-overfitting.
2. בשלב הבא נרצה לקבוע קצב למידה. נסתכל על ה-loss תוך כדי תהליך הלמידה, אם הוא לא יורד נגדיל את קצב הלמידה ואם עולה נוריד. נבחר את קצב הלמידה שגורם ל-loss לרדת משמעותית תוך 100 איטרציות. בדרך כלל קצב הלמידה יהיה באזור אחד מהערכים: 1e-1, 1e-2, 1e-3, 1e-4, וה-decay rate באזור 1e-4, 1e-5, 0.
3. נסתכל על עקומת הדיוק על ה-train וה-validation לבדוק אם יש overfitting או underfitting. נטפל בבעיות אלו כמתואר למטה ונחזור על שלב זה. אם הדיוק ממשיך לעלות גם ב-train וגם ב-validation נגדיל את מספר ה-epochs כדי לאמן את המודל יותר.
4. נרצה להשוות בין מספר סוגי אופטימייזרים, ארכיטקטורות שכבות שונות, ורגולריזציות כדי שנוכל לבחור את המודל הטוב ביותר.

### Overfitting

Overfitting הוא התאמה של המודל ל-train אך לא למידע אחר. יש לכך מספר פתרונות:

* רגולריזציה - באמצעות lasso, ridge או dropout.
* Early stopping - לעצור לפני שמגיעים ל-overfitting.
* לפשט את המודל על ידי הורדת שכבות ונוירונים. באופן עקרוני, רשת עם מעט נוירונים יהיה בה פחות overfitting אך יהיה יותר קשה לאמן בצורה טובה.
* לבחור היפר-פרמטרים אחרים.
* הוספת דאטה מייצג כך שמודל ילמד טוב יותר.

### Underfitting

התוצאות על ה-train לא טובות. יש שני סוגים של פתרונות:

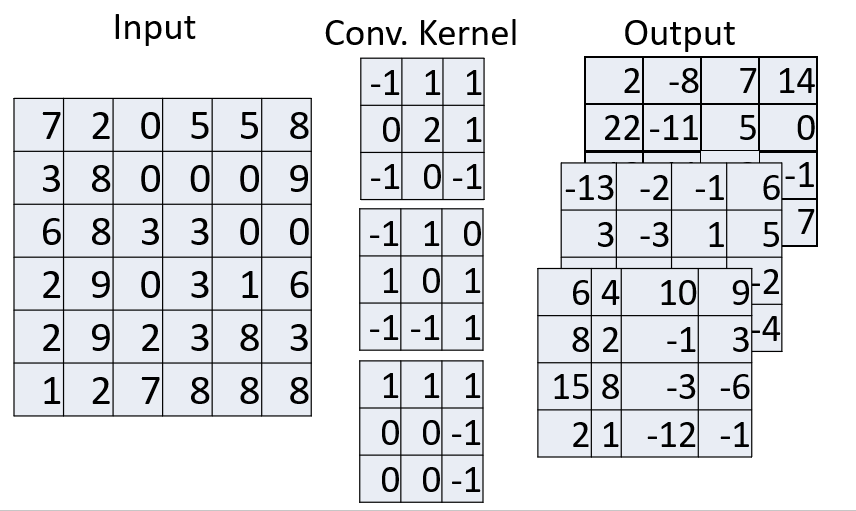
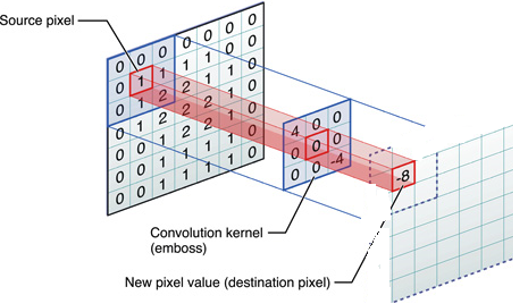
* לאמן יותר צעדים.
* להוסיף שכבות
* לשנות קצב למידה ו-optimizer.

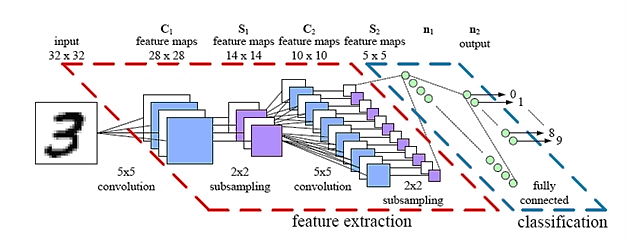
# Convolutional Neural Networks

## הגדרה

עד כה עסקנו בשכבות שהן Fully Connected, כלומר כל נוירון (תכונה) הוא צירוף ליניארי של כל הנוירונים בשכבה הקודמת עם הפעלת פונקציית אקטיבציה. בשיטה זו אנו לא מייחסים חשיבות לתכונות קרובות ובנוסף משתמשים בהמון משקלים. ב-Convolutional network (רשת מפותלת) כן מייחסים חשיבות לתכונות קרובות. כיוון שבדאטה המכיל תמונות יש חשיבות גדולה לקרבה בין התכונות (פיקסלים), רשתות מסוג זה מאוד מתאימות לחיזוי על תמונות. הרעיון הכללי ב-Convolutional network מאוד דומה לאיך שהמוח מעבד תמונה. שיטה זו הופכת את המודל למורכב אך בו בזמן לא משתמשת בהרבה משקלים כמו ברשת שהיא fully Connected.

הרעיון המרכזי הוא שבכל שכבה יש K פילטרים, הנקראים גם גרעינים (kernels). כל פילטר הוא מטריצה בגדול . כל פילטר מכפילים בכל חלק בתמונה משכבה קודמת, החל מהקצה העליון השמאלי ובקפיצות (stride) בגודל S ימינה ולמטה (Sliding Window). הכפלת הפילטר בחלק מהתמונה אינה הכפלת מטריצות אלא הכפלה בין איברים מקבילים וסכימה של כל ההכפלות (elementwise). ניתן גם לרפד את התמונה משכבה קודמת במסגרת בגודל P של אפסים. מרפדים ב-0 כדי לשלוט בגודל התמונה הנוצרת מכל פילטר. אם התמונה משכבה קודמת היא בגודל ורוחב , אזי מכל פילטר נוצרת תמונה חדשה בגודל , כאשר



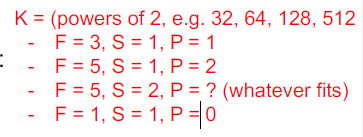
על כל ערך שנוצר מהכפלת פילטר בחלק מהתמונה מפעילים פונקציית אקטיבציה, כמו ReLU. אפשר גם להוסיף bias. התמונה הסופית שמתקבלת מכל פילטר לאחר האקטיבציה נקראת activation map. כל activation map מייצגת תכונה כלשהי על התמונה המקורית. בכל שכבה חוזרים על תהליך זה עם קבוצת פילטרים אחרים, כך שמקבלים activation map קטנים יותר ויותר בכל שכבה. כל ה- activation mapsבשכבה מסוימת נקראים convolutional layer. השכבה האחרונה ב-CNN היא fully connected כי רוצים להתחשב בכל התכונות המיוחדות שמצאנו בתהליך הלמידה. על גבי השכבה האחרונה מפעילים Sigmoid או Softmax כדי לקבל את החיזוי הסופי.

כאמור, כל activation map שנוצרת מהכפלת הפילטרים מייצגת תכונה כלשהי על התמונה המקורית. ככל שה- activation map בשכבה עמוקה יותר היא מורכבת מיותר פיקסלים מהתמונה המקורית ולכן מסוגלת לבטא תכונות כלליות יותר על התמונה. activation map בשכבה נמוכה מייצגת תכונה מקומית על חלק מסוים של התמונה.

לסיכום, כל שכבה ב-CNN יש ארבעה היפר-פרמטרים:

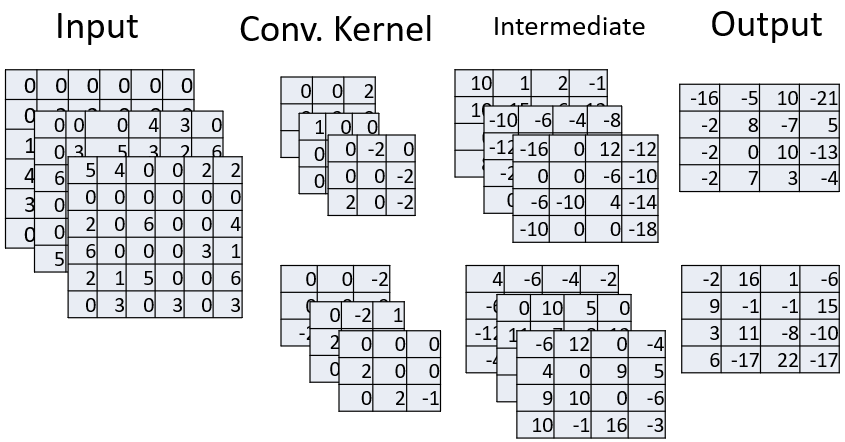
* מספר הפילטרים K.
* גודל הפילטר F.
* גודל הצעד S.
* גודל הריפוד באפסים P.

אם הקלט לשכבה זו הוא C תמונות שכל אחת ברוחב ואורך , כלומר קלט בגודל , אזי הפלט הוא בגודל , כאשר . מספר המשקלים בשכבה זו הוא .



משמאל יש מספר ערכים נפוצים להיפר-פרמטרים אלו.

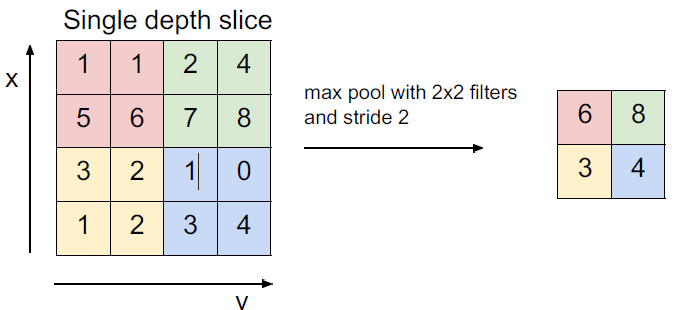
## 4-D Convolution

שיטה נוספת לביצוע קונבולוציה היא שבמקום להגדיר K פילטרים שמפעילים כל אחד מהם על כל אחד מ-C התמונות משכבה קודמת, מגדירים T קבוצות של C פילטרים. לכל קבוצה מפעילים את כל הפילטרים בקבוצה על התמונות משכבה קודמת בהתאמה, כך שמתקבל K תמונות שסוכמים את כולם כדי לקבל תמונה אחת שהיא ה-activation map לשכבה הבאה. בסך הכל הפלט לשכבה הבאה הוא T activation maps.

## Pooling Layer

לאחר כל שכבת קונבולוציה ניתן להוסיף שכבה של pooling. בשכבה זו מחלקים כל אחד מהתמונות בשכבה קודמת למרובעים בגודל F. על כל הערכים בריבוע מפעילים פונקציה כלשהי (נפרט בהמשך) כדי לקבל ערך אחד. הערכים בכל ריבוע יוצרים תמונה חדשה בגודל המועברת כפלט לשכבה הבאה. יש שני סוגי פונקציות לשכבה זו:

Max pooling - מחזירים את הערך המקסימלי מכל ריבוע (ראה תמונה). משתמשים בדר"כ בפונקציה זו.

Average pooling - מחזירים את ממוצע כל הערכים בריבוע.

הרעיון מאחורי שכבה זו הוא בעיקר לדלל את המידע ולקחת רק את החלקים החשובים שבאמת צריך בשביל הסיווג. דילול והפחתת המידע גורמת לכך שצריך פחות פרמטרים בשכבות הבאות וכך יהיה קל יותר לבצע בהן קונבולוציה. כיום פחות משתמשים ב-pooling כדי להפחית מידע ופשוט בוחרים stride ופילטר גדולים יותר.

## Transfer Learning

עד כה כל האלגוריתמים של למידה שראינו בנו מודל מאפס באמצעות דאטה. ב-Transfer Learning לעומת זאת, מנצלים מודל קיים שאומן היטב על בעיה אחת כדי לפתור בעיה אחרת דומה. אנו בעצם מנצלים את יכולות הלמידה של המודל המאומן כדי לפתור בעיות דומות. אסטרטגיה זאת מאפשרת לנו להשתמש במודל הידוע כאיכותי בלי שהשקענו את המאמצים הרבים הכרוכים בלאמן אותו.

יש צורך לאמן את השכבות האחרונות כדי להסיק מסקנות הנוגעות לבעיה החדשה מכל התכונות שגילה המודל.

* אם הדאטה בבעיה החדשה הוא קטן יחסית ודומה לדאטה שעליו התאמן המודל, מספיק לאמן רק את השכבה האחרונה שהיא FC שתחזיר את מספר המחלקות הרצוי בבעיה החדשה. אם הדאטה אינו דומה, יש לאמן מספר שכבות שונות.
* אם הדאטה בבעיה החדשה הוא גדול ודומה לדאטה שעליו המודל התאמן, יש לאמן מספר שכבות אחרונות כדי להסיק מסקנות איכותיות. מומלץ שהקצב למידה יהיה נמוך מקצב הלמידה המקורי. ככל שהדאטה החדש שונה מהדאטה שעליו התאמן המודל יש להגדיל את מספר השכבות שאנו מאמנים בבעיה החדשה.

## ארכיטקטורות CNN

נסקור את הארכיטקטורות המקובלות של CNN שהתפתחו במהלך השנים, החל משנת 2014. נתאר כל ארכיטקטורה, מה נתחדש בה, ונסביר מה היתרונות והחסרונות שלה.

AlexNet – הארכיטקטורה הראשונה שהביאה לפריצה בתחום זה. כוללת בסך הכל 8 שכבות של קונבולוציה עם max pooling שבסופם שלוש שכבות Fully Connected. בארכיטקטורה זאת היו המון פילטרים גדולים כדי לצמצם את הדאטה. צריך לזכור שבזמן זה היה מאוד קשה להריץ CNN והיה צריך לצמצם את הדאטה מהר.

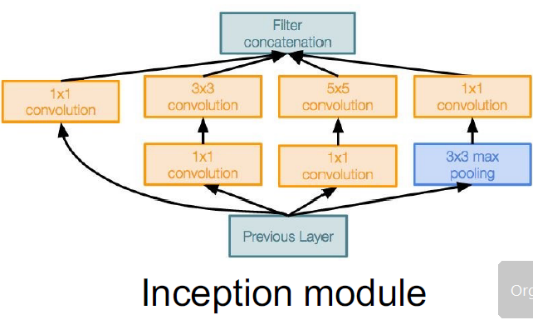
ZFNet – מאוד דומה ל-AlexNet, אלא שהשתמש ביותר פילטרים שהם קטנים יותר.

VGGNet – כאן השתמשו ב-16-19 שכבות. החידוש המרכזי שהקטינו מאוד את הפילטרים שלא יהיו גדולים יותר מ-. כפיצוי העמיקו מאוד את הרשת כדי שבשכבות העמוקות הפילטרים יהיו משמעותיים יותר והתחשבו ביותר פיקסלים של התמונה המקורית. ברשת זו היו 138 מיליון משקלים.

GoogLeNet – כאן השתמשו ב-22 שכבות. החידוש המרכזי היה בארכיטקטורה זו הוא ה-“Inception module”, שלפיו בכל שכבה היה ניתן לבחור איזה גודל פילטר רוצים בשכבה הבאה. שיטה זו מאפשרת להוריד את מספר הפרמטרים משמעותית.

ResNet –

Efficient Network -

מסתבר שמודלים אלו, במיוחד המאוחרים שבהן, מאוד טובים בלמידת תכונות משמעותיות על תמונות. לכן כיום מקובל לקחת ארכיטקטורות אלו, לאחר שלב האימון, ולהשתמש בהן על בעיות אחרות. צריך לבצע התאמות בשכבות האחרונות שהן Fully Connected כדי להתאים את המודל לדאטה של הבעיה החדשה.