למידת מכונה

חיזוי סדרת זמן בעזרת מודל LSTM

נוראל גליק

שם בית הספר: הכפר הירוק ע"ש לוי אשכול

סמל המוסד: 580019

כתבת: כפר הירוק 47800 רמת השרון

טלפון: 03-6455621

שם התלמיד: נוראל גליק

מספר ת.ז: 336266499

כתובת: שער הים 21 הרצליה

נייד: 054-9116102

כתובת מייל: norelglick21@gmail.com

בית הספר: הכפר הירוק ע"ש לוי אשכול

מספר סמל בית הספר: 580019

פרטי המנחה:

שם המנחה האקדמי: זוהר ברונפמן

מספר ת.ז: 038063368

כתובת: Sapir Tower, Tuval 40, Ramat Gan, 52522

נייד: 058-755599

כתובת מייל: zohar@pecan.ai

תואר אקדמי: PhD Computational Cognitive Neuroscience

מקום עבודה: Pecan.ai

Contents

מבוא

בשנים האחרונות למידת מכונה נמצאת כמעט בכל מקום. מהתוכנת צילום בטלפונים הניידים שמתקנת את מצב התאורה בתמונה, עד לחברת פרסומות שבוחרת איזו פרסומת תוצג ברחוב ובטלוויזיה. כיום, קיימות אינסוף סיבות להכניס למידת מכונה לזרימת עבודה של חברות, ואפילו לחיי היום יום שלנו.

אחת מהאבני הפינה של כלכלה מודרנית היא שוק ההון. בשוק ההון חברות יכולות להנפיק את עצמם לציבור ולגייס כסף. בנוסף אנשים יכולים לנסות להתעסק בשוק ספציפי ולהרוויח כסף. כמו כן, שוק ההון מייצג את הכלכלה של המדינה. ומי שיוכל לחזות את שוק ההון, גם רק שוק אחד ספציפי, יכול לעשות מיליונים.

אז, אם למידת מכונה היא כל כך פופולרית וחזקה בשנים האחרונות, האם היא יכולה לעזור לנו בשוק ההון? זוהי השאלת חקר שלי לעבודה זאת. הדגש בעבודה יהיה על למידת מכונה ולא על שוק ההון. הסיבה לכך היא שמראש התשובה שנמצא בסוף העבודה היא שאי אפשר לחזות מחירי מניות. אם היה אפשר לחזות מחירי מניות עם הטכנולוגיה של היום, כמות גדולה של אנשים היו יכולים להתעשר משוק ההון , ושוק ההון היה קורס. לכן אני אומר מראש: הנושא המרכזי בעבודה זאת הוא על למידת מכונה מודל ה-LSTM, והתמודדותו עם סדרות זמן. לכן בחרתי במניות, כי מניה היא סדרת זמן שתלויה באינסוף פרמטרים ולכן ישמש כStress Test טוב לניסוי שלי.

הציפיות שלי מהעבודה היא שהעבודה תלמד אותי ואת הקורא על למידת מכונה ברמה המתמטית. זה נושא שהוא קשה ללמוד לבד, וגם אחר עבודת חקר קטנה אי אפשר להיות מומחה בנושא. אבל אני מצפה שאני והקורא נפתח את ראשינו לנושא זה ושנפסיק לחשוב על למידת מכונה כ"קופסא שחורה". אחד מהמטרות שלי לעבודה זו הוא להסביר כמה שיותר טוב את המתמטיקה מאחורי למידת מכונה בלי שהקורא יצטרך לדעת נושאים שלמים של אלגברה לינארית או חשבון דיפרנציאלי ואינטגרלי רב משתנים.

בינה מלאכותית

בינה מלאכותית נשמעת כמו קסם. איך בני אדם יכולים להכין בצורה מלאכותית משהו עם אינטליגנציה? התשובה היא שאי אפשר. למרות השם, בינה מלאכותית לא מתייחסת למחשב עם מוח ואינטליגנציה כמו שיש לבני אדם, אלה למערכת המורכבת מפונקציות מתמטיות שמטרתם היא ללמוד.

מכיוון שבני אדם חשופים לכמות מידע אינסופית לאורך חייהם והמוח שלנו הוא מערכת מורכבת מאוד שאחד מהמטרות שלה הוא ללמוד, לאורך החיים אנחנו לומדים עוד ועוד, ויכולים לפעול בצורה שונה עבור כל דברים שונים שאנחנו נתקלים בהם. זה ההתנהגות שבינה מלאכותית מנסה לחקות.

בפרויקט בינה מלאכותית יש שלושה שלבים: שלב החקירה, שלב הדאטא(data), ושלב בחירת המודלים. כמו שבני אדם מקבלים דאטא מהעולם שמסביבם, כך גם מודלים של למידת מכונה צריכים דאטא. השלב הראשון, שלב החקירה, עוסק בלהבין את מטרת הפרויקט (מה אנחנו רוצים לחזות) ואת העולם שלו. לעבודה שלנו המטרה היא לחזות מחיר של מניה בעתיד, ולכן בשלב החקירה צריך לחקור על עולם המניות: מה היא מניה, וממה היא מושפעת. השלב השני, שלב הדאטא, עוסק בלמצוא את הדאטא הזה ולסדר אותה בצורה הכי טובה למודל שלנו. ארחיב על זה בהמשך העבודה. השלב השלישי, שלב בחירת המודלים, מטרת שלב זה הוא לנסות את הדאטא שלנו על הרבה סוגי מודלים עם פרמטרים שונים. פרמטרים אלו יכולים להיות קצב הלמידה של המודל, כמה דפוסים הוא יכול ללמוד וכמה עמוק הוא יכול להבין את הדפוסים שימצא ועוד.

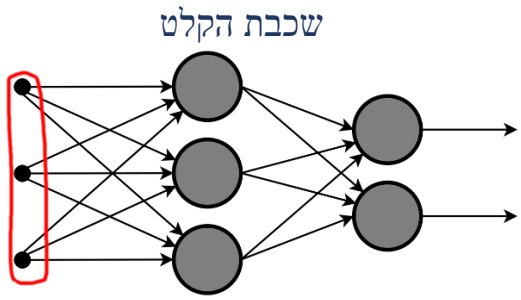
אפשר להסיק מהפסקה הקודמת שקיימות הרבה בעיות עם בינה מלאכותית ושהמערכת הזאת צריכה בני אדם שיטפלו בה בצורה קפידה כדי שהיא תצליח לעשות משהו שלבני אדם נראה מאוד אינטואיטיבי וקל. לכן השם מטעה, ואין פה באמת אינטליגנציה. אולי בעוד מאה שנה, בני אדם יצליחו להכין מערכת שלא תלויה בהם. כעט, נלמד על רשת נוירונים ולאט לאט נרחיב על המערכת הנפלאה שנקראת בינה מלאכותית.

רשת נוירונים

עכשיו שאתם יודעים מה היא בינה מלאכותית, נכנס לתוך הנושא של למידה עמוקה. בעקרון מה שמבדיל בין למידת מכונה ללמידה עמוקה היא הסגנון שמשתמשים בו כדאי לבצע את הבינה המלאכותית. למידת מכונה משתמשת באלגוריתמים יחסית בסיסיים כדי ללמד את המחשב לחזות פרמטר מסוים. דוגמא לכך היא אלגוריתם ה"רגרסיה לינארית" (Linear Regression). אמנם המודלים המורכבים של למידה עמוקה נחשבים מודלים של למידת מכונה, אך נהוג לעשות ביניהם הפרדה.

אז, מה הוא מודל של למידה עמוקה? מודל של למידה עמוקה יכול ללמוד דפוסים מאוד מורכבים. לדוגמא, ניקח מודל של "Fully Connected Layer" (FC Layer), או בשמו המוכר יותר, "Neural Network" (רשת נוירונים). רשת נוירונים הוא מרכיב מאוד יסודי בעולם הלמידה עמוקה, וכך היא עובדת:

השכבות ברשת נוירונים



כל רשת נוירונים מורכבת ממספר שכבות, שלכל שכבה תפקיד אחר. השכבה הראשונה נקראת "שכבת הקלט" (Input Layer). השכבה הזאת משמשת רק כייצוג לקלט שנכנס לרשת הנוירונים. בספריות הפופולריות (כמו TensorFlow) הקלט הוא רשימה המחזיקה עוד רשימות שבהן יש את המידע שניתן להעביר לרשת נוירונים. לדוגמא, נראה את הקלט של הדאטא סט המפורסם MNIST:

```
import tensorflow.keras.datasets.mnist as mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

print(x_train.shape)
print(y_train.shape)
```

```
(60000, 28, 28)
(60000,)
```

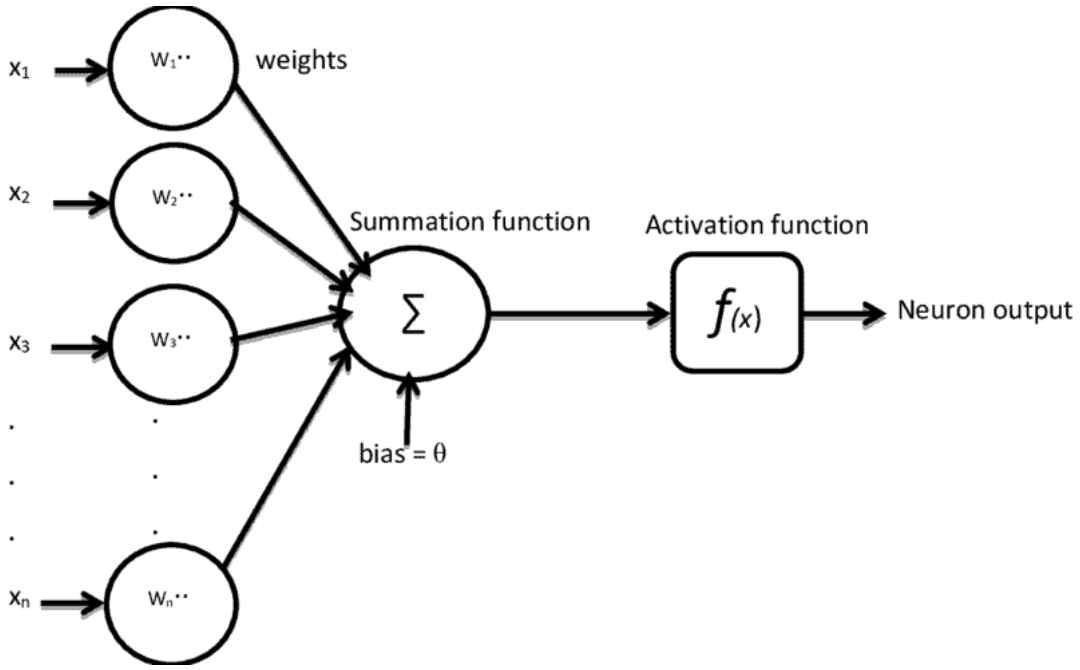
x_train היא רשימה המחזיקה בתוכה 60,000 רשימות אחרות שבכל אחת מהן יש ייצוג של תמונה בעל גודל של 28 פיקסלים על 28 פיקסלים. נהוג לקרוא לנתון אחד במידע "דגימה". דוגמא לדגימה היא:

בתמונה הראשונה הנוירונים שמסומנים באדום הם הפיקסלים המרכיבים את התמונה של ה-5. מכיוון שהתמונה היא 28x28, כמות הנוירונים שיהיה הוא 784. נהוג לקרוא לזה "כמות הfeatures במודל". y_train הוא רשימה באורך 60,000 שמחזיקה את התווית של כל תמונה. לדוגמא, ל-5 הזה יהיה תווית של "5". כאשר מלמדים את המודל, נותנים לו את התווית וכך הוא יודע לתקן את עצמו.

השכבה הנסתרת

השכבה הנסתרת היא ה"מוח" של המודל. תפקידה ללמוד דפוסים במידע שניתן לה משכבת הקלט. השכבה מורכבת ממספר נוירונים שמוגדר מראש, וכל נוירון תפקידו ללמוד דפוס אחר. אבל מה זה בכלל נוירון? נוירון הוא "אלגוריתם" (אפשר לחשוב על אובייקט שבו יושב משתנים ופונקציות) שיש לו תכונות: משקולות, bias, ופונקציית Activation:

* התמונה מייצגת את האלגוריתם של נוירון אחד



בתמונה אפשר לראות את כל הפעולות שנוירון אחד עושה:

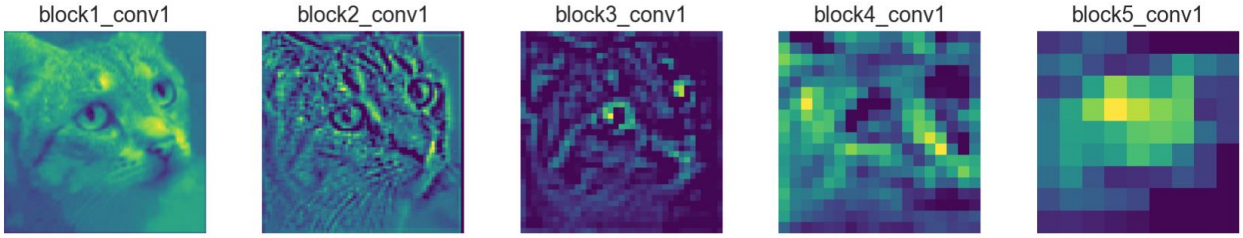
1) הוא מכפיל כל משקולת עם הדגימה המיוחסת אליו (הנוירונים משכבת הקלט)
2) הוא עושה סכום של כל ההכפלות מהצעד הקודם ומוסיף bias. נלמד על זה בהמשך.
3) הוא מכניס את הסכום הזה לפונקציית Activation ואנחנו מקבלים את התוצאה של הנוירון.

שלבים 1 ו-2 הם ביחד הצירוף הלינארי של המשקולות עם כל דגימה. מפה והלה, אשתמש במושג ה "צירוף הלינארי" הרבה אז תזכרו שזה פשוט הכפלת המשקולות עם הדגימות המיוחסת אליהם והסכמה של כל ההכפלות האלה.

לפני שאנחנו מאמנים את המודל, המשקולות האלה מאותחלות כמספר רנדומלי. קיימים הרבה סוגים של איתחולים של משקולות וזה מאוד תלוי בסוג הפרדיקציה שרוצים לעשות.

כאשר יש יותר משכבה נסתרת אחת, המודל נהיה מודל של "Deep Learning" מכיוון שהשכבות הבאות לומדות דפוסים שמתבססים על הדפוסים הנלמדו בשכבות קודמות, אפשר ללמוד דפוסים מאוד מורכבים.

לדוגמא, ניקח את התמונה הבאה:



*בתמונה הספציפית הזאת השתמשו במודל CNN, התמונה להמחשה בלבד

התמונה הראשונה (משמאל) היא התמונה המקורית. התמונה השנייה היא התוצאה של השכבה הנסתרת הראשונה, והיא מדגישה outlines. התמונה השלישית היא התוצאה של התוצאה הנסתרת השנייה והיא מדגישה את העיניים, וכך הלאה... כפי שאפשר לראות, בכל שכבה יש כמות פיקסלים יותר נמוכה וזה כדי שהמודל ילמד features קטנים אך רבים, ולכן גם כמות הנוירונים עולה בכל שכבה (זה לא בהכרח נכון לכל מודל! זה תלוי ביוצר המודל ותוצאותיו אחרי הרבה ניסוי וטעיה).

ככל שיש יותר שכבות נסתרות עם יותר נוירונים, כך התוצאות של המודל יהיו יותר טובות (לדוגמא: הסיכויים שהמודל יזהה חתול יעלה). אבל תמיד צריך יחס טוב! אם יש לך כמות דאטא מאוד קטנה ומודל מאוד מורכב, התוצאות יהיו לא טובות. זה נכון גם בהפוך, אם יש לך כמות דאטא מאוד גדולה ומודל מאוד פשוט. ML Engineers חלק מעבודתם להבין מה הם יכולים לבנות עם כמות הדאטא שיש להם, ואם צריך עוד דאטא או לא.

שכבת הפלט

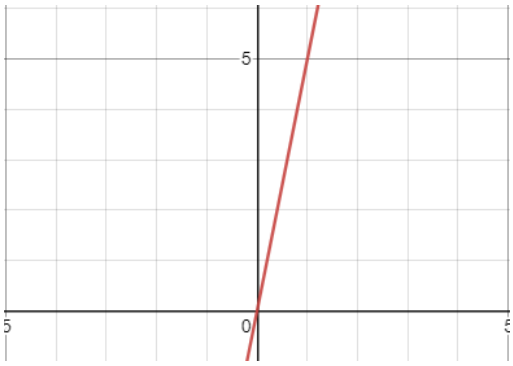
שכבה זו עושה קומבינציה של התוצאות של השכבה הנסתרת (בשמו המתמטי נקרא צירוף לינארי), נכנס לפונקציית Activation מסוימת (שיוצר המודל בוחר), ויוצא פרדיקציה/פלט.

אימון הרשת

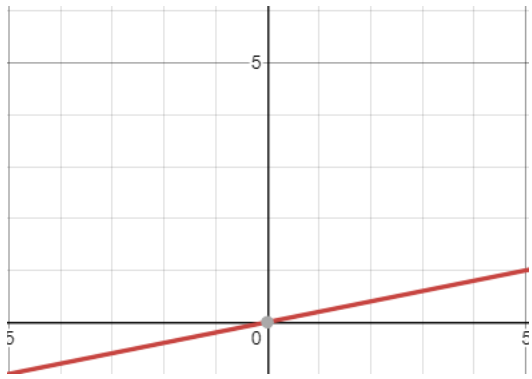
כאשר אנחנו רוצים לאמן את המודל שלנו, אנחנו מביאים לו דגימה (שכבת הקלט) והוא מחשב פרדיקציה, ובנוסף מביאים לו את הפרדיקציה שהייתה אמורה להתקבל (התווית). לדוגמא נביא למודל שמטרתו לנבא את הפונקציה: $y(x) = 5x$ את הערך 2 והוא יעשה פרדיקציה של 9. אנחנו מבינים שיש כאן בעיה, מכיוון שידוע לנו שהפרדיקציה הייתה צריכה לצאת 10. לכן, באימון המודל, אנחנו מביאים לו את הדגימה ובנוסף את התווית. המודל משתמש בתשובה הזאת בפונקציית ההפסד:

פונקציית ההפסד היא פונקציה שמחשבת לנו כמה המודל פספס את המטרה שלו בפרדיקציה. ישנם כל מיני סוגים של פונקציות הפסד, ואיזה אחד אתה בוחר לאלגוריתם הלמידה שלך מאוד תלוי במטרת המודל. לדוגמא, מישהו שבונה מודל שמטרתו לעשות פרדיקציה על תמונה כדי לדעת אם התמונה היא תמונה של כלב או חתול ישמש בפונקציית ההפסד Binary Cross-Entropy, מכיוון שהוא יכול להגיד למודל בכמה אחוז הוא נבא חתול ולא כלב.

המטרה שלנו הוא להראות למודל כמה הוא טעה בפרדיקציה, ולבנות פונקציה שתשתמש בטעות הזו כדי שבפעם הבאה שהמודל יקבל את אותו הדגימה הוא יחשב את התשובה הנכונה. כדי להסביר, נשתמש בפונקציה מאוד פשוטה שנרצה להגיע אליה: $y(x) = 5x$.

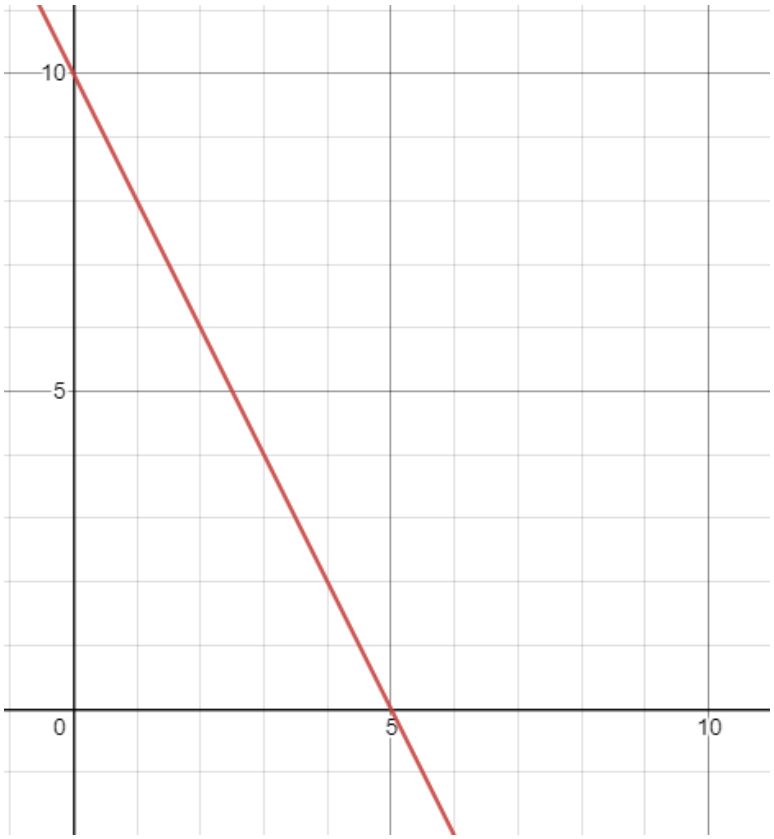


נכין מודל פשוט: $g(x) = 0.2 * x$. (0.2 הוא המשקולת w, שהותחל כמספר רנדומלי)



נביא לו דגימה ותווית $(2,10)$ מכיוון ש: $f(2) = 5 * 2 = 10$, ונחשב את ההפסד:

$g(2) = 0.2 * 2 = 0.4 \rightarrow l = 10 - 0.4 = 9.6$. קיבלנו הפסד של 9.6. אנחנו רוצים לשאוף להפסד של אפס, לכן זה ממש גרוע ואנחנו יודעים שהמודל שלנו במצב רע. אז, איך אפשר לשנות את w כך שנקבל הפסד של 0? אם נסתכל על גרף של ההפסד כתלות בערך של w (שבו ציר הY הוא ההפסד וציר הX הוא ערך w) בכך שנציב את התווית נקבל: $f(w) = 10 - (2 * w)$



לפני שנתבונן בגרף, נבין שהפונקציה $f(w)$ מסמלת את כל הערכים האפשריים לw ואיך המודל שלנו יגיב. לדוגמא, נכניס $w = 2$ ונקבל: $f(2) = 10 - (2 * 2) = 10 - 4 = 6$. עכשיו, אם נציב במודל שלנו

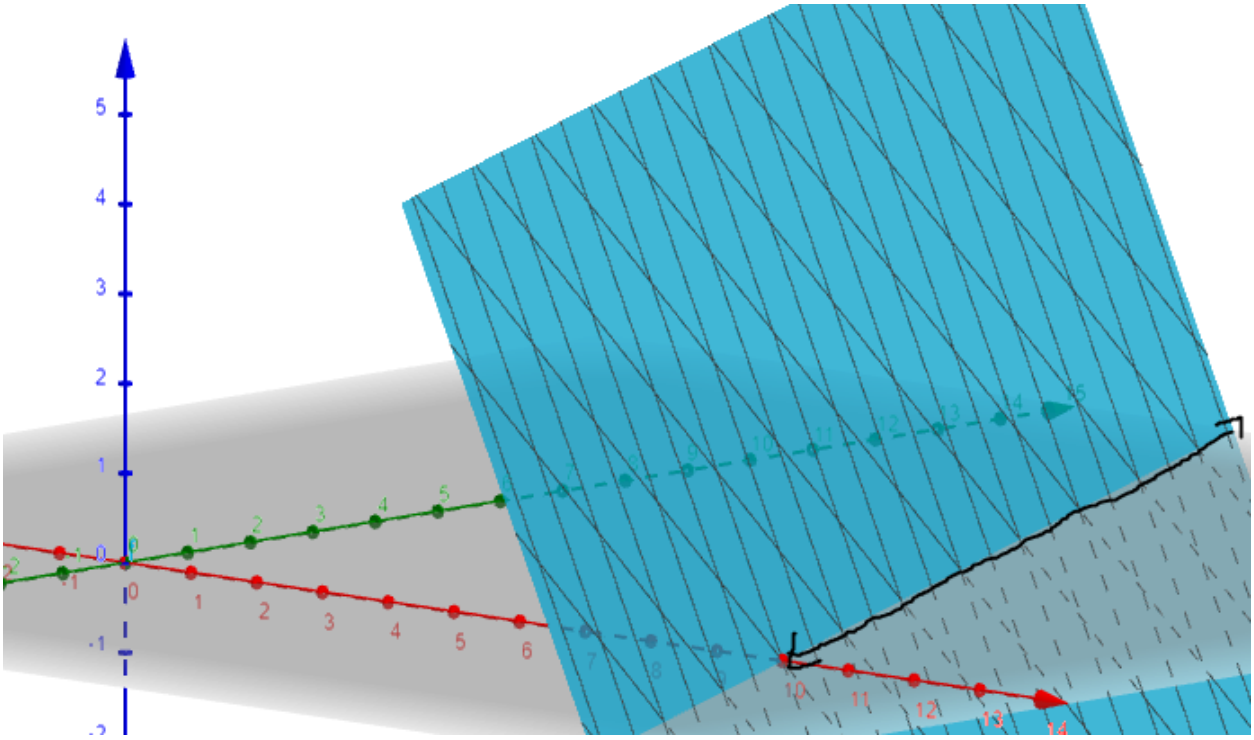
$w = 2$ נראה ש: $g(2) = 2 * 2 = 4$. נציב בפונקציית ההפסד: $l = 10 - 4 = 6$. הפונקציה $f(w)$ מראה לנו מה תהיה ההפסד לפי איזו w נציב בה. זה מצוין, מכיוון שאפשר עכשיו לחשב בעזרת אלגברה מה תהיה w אם אנחנו רוצים לקבל תשובה של 0. נעשה זאת בכך שנציב: $f(w) = 0$ ונפטור:

$$f(w) = 0 \rightarrow 10 - (2 * w) = 0 \rightarrow \ -2 * w = -10 \rightarrow w = 5$$

יצא לנו $w = 5$. נתבונן בגרף ונראה שהגרף מראה זאת גם. אם נציב את זה במודל שלנו, נקבל את המודל הבא: $g(x) = 5x$. זה בדיוק שווה ל-$f(x)$. המסקנה מפה הוא שכדי למצוא את הערך הכי אופטימלי למודל שלנו, צריך להכין פונקציה שתדמה את ההפסד כתלות במשקולת. בפונקציות ליניאריות דבר זה הוא טריוויאלי, בדיוק כמו שעשינו. אבל מה קורה כאשר יש לנו מודל ממש מורכב, והוא אפילו לא בשני ממדים, אלה שלוש, ארבע, ואפילו מיליארד? היינו גוזרים כדי למצוא את המינימום. פה נכנס האלגוריתם של Gradient Descent, אליו בנוי Backpropagation:

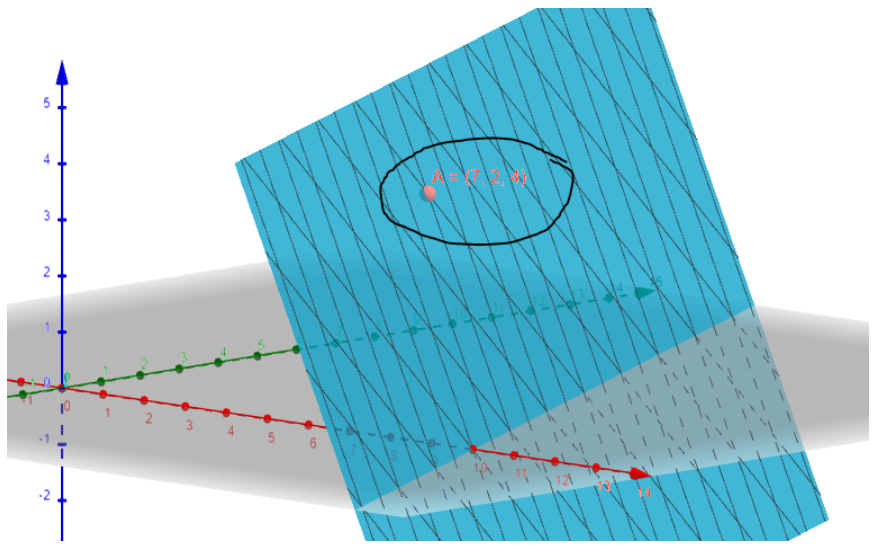
Gradient Descent

ניקח את המודל $g(x, y) = x * w_x + y * w_y$, פונקציית ההפסד $l = y_{label} - y_{pred}$, ותווית $(2, 1, 20)$. כמו לפני, נכין פונקציה של ההפסד כתלות במשקולות ונקבל: $f(w_x, w_y) = 20 - (2 * w_x + 1 * w_y)$



זה הגרף של $f(w_x, w_y)$. קצת יותר מסובך לא? אמנם אפשר בעזרת אלגברה למצוא את המשקולות האופטימליות, מה יקרה כאשר יש לנו מיליארד משקולות, ומודל מאוד מאוד מסובך? המחשב לא יצליח לחשב את המשקולות האלה בזמן שהוא הגיוני. לכן, Gradient Descent קיים, ומטרתו להצליח למצוא, בקירוב, את המשקולות האופטימליות גם אם קיימות הרבה מאוד משקולות.

Gradient Descent היא פונקציה איטרטיבית, כלומר קיים אלגוריתם מסוים וממשיכים להריץ את האלגוריתם הזה עד שנקבל תוצאה שאנחנו שמחים איתה. הדרך שבה Gradient Descent מצליח למצוא את המשקולות האופטימליות הוא בעזרת הנגזרות החלקיות (Gradients) של פונקציית ההפסד. האלגוריתם מחשב את הנגזרות החלקיות, מכפיל אותם (לפי כלל השרשרת שקיים בפונקציות רב ממדיות) ומוריד מהערך של המשקולת. נראה איך זה עובד בזמן אמת:



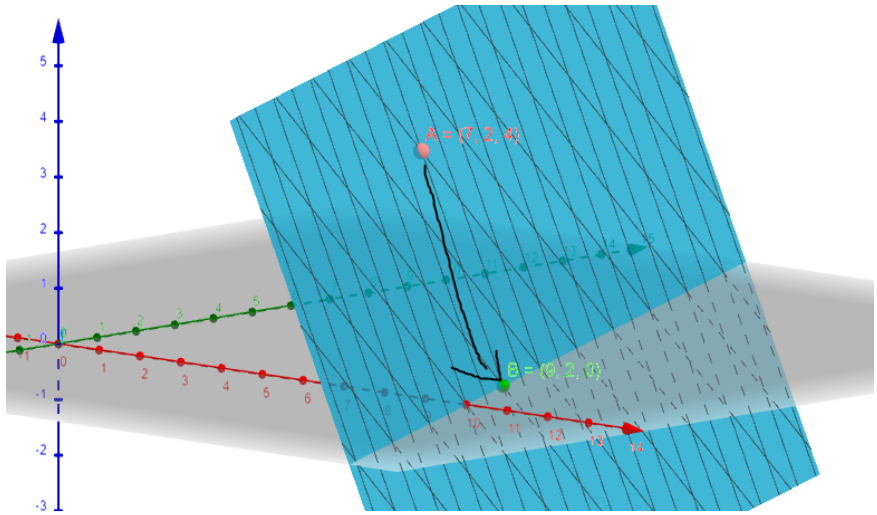
נכניס נקודה (7,2,4) על פונקציית ההפסד. כלומר, $w_x = 7, w_y = 2, l = 4$. כדי לדעת איך להגיע ל-0, צריך לגזור את פונקציית ההפסד מכיוון שזה ייתן לנו את הכיוון ל-0. נגזור:

$$\frac{df}{dw_x} = 20 - \left(2 * w_x + 1 * w_y\right) = 0 - 2 * w_x - 0 = -2$$

קיבלנו -2, ולכן צריך להחסיר מw_x את ערך זה ונקבל $w_x = 9$. נציב ב$f(w_x, w_y)$ ונקבל:

$$f(9, 2) = 20 - (9 * 2 + 1 * 2) = 20 - (18 + 20) = 20 - 20 = 0$$

קיבלנו 0! זה אומר שהמשקולות האלה הם הכי אופטימליות בשביל הדגם מידע (2, 1, 20).



אפשר להתבונן בגרף ולראות את כל הערכים שפוגעים בציר ה-X. כל הערכים האלו הם ערכים אופטימליים למודל שלנו.

למדנו כעט בצורה פשוטה מאוד מה האלגוריתם הכי בסיסי בעולם הלמידה עמוקה עושה. נלמד עכשיו על Backpropagation, האלגוריתם שמשלב רשת נוירונים מלאה עם Gradient Descent!

Backpropagation

האלגוריתם הכי משומש לאמון רשת נוירונים היא Backpropagation. אלגוריתם זה משתנה בין מודל למודל ובין רשת לרשת (נבין בהמשך למה).

אלגוריתם הBackpropagation:

- המודל שלנו יהיה רשת נוירונים עם שכבת קלט, שכבה נסתרת אחת שבה יש נוירון אחד, ושכבת פלט. הקלט יהיה מספר אחד (1 feature), פונקציית הActivation שלנו תהיה Sigmoid, ופונקציית ההפסד שלנו תהיה $L(x) = y - y_{pred}$ כאשר y_{pred} הוא הפרדיקציה של המודל וy הוא התווית. המשקולות במודל יהיו מאותחלות כמספר רנדומלי בין -1 ל1

1) Forwardpropagation: בשלב הזה המודל שלנו לוקח את כל הדאטא סט ,עושה על כל דגימה פרדיקציה, ומחשב את ההפסד של אותה דגימה בעזרת פונקציית ההפסד והתווית.

$$l(x_i, y_i) = y_i - \sigma(x_i * w_i + b)$$

2) חישוב ההפסד: בסיום הForwardpropagation, עושים ממוצע לכל ההפסדים של כל הדגימות כדי להגיע למספר אחיד.

$$L = \frac{1}{n}\sum_{i=0}^{n} l(x_i, y_i)$$

3) חישוב הנגזרות החלקיות: כעט צריך לחשב את הנגזרות החלקיות של הפונקציות הבונות את המודל. תמיד צריך לגזור בכוונה להגיע למשקולת שאנחנו רוצים לעדכן (במקרה שלנו w_i). הדרך שנעשה את זה הוא כמו לקלף בצל – נתחיל מהחלק העליון ולאט לאט נגיע לאמצע. השלב הזה הוא הכי קשה, ולכן אפרק את החלק הזה לכמה חלקים:

a. הנגזרת של פונקציית ההפסד. מכיוון שהמשקולת שלנו נמצאת ב y_{pred}, נגזור לפיו:

$$l(y_{pred}, y_{label}) = y_{label} - y_{pred} \rightarrow \frac{dl}{dy_{pred}} = -1$$

b. הנגזרת של פונקציית Sigmoid. לפני פונקציית ההפסד, אנחנו מכניסים את $x_i * w_i + b$ ל-Sigmoid ולכן זוהי הגזירה הבאה שלנו:

$$\frac{d\sigma}{dx} = \sigma(x) * [1 - \sigma(x)] = \sigma(x_i * w_i + b) * [1 - \sigma(x_i * w_i + b)]$$

c. הנגזרת של $x_i * w_i + b$. זאת הנגזרת האחרונה:

$$\frac{d(x_i * w_i + b)}{dx_i} = w_i$$

d. נכפיל את כל הנגזרות החלקיות שחישבנו ונקבל את Δl:

$$\Delta l = -1 * \sigma(x_i * w_i + b) * [1 - \sigma(x_i * w_i + b)] * w_i$$

e. נכפיל את Δl בlearning rate α שנבחר כדי להחליט באופן מלאכותי מה תהיה השפעתו, ונסיר מw_i כדי לעדכן אותו:

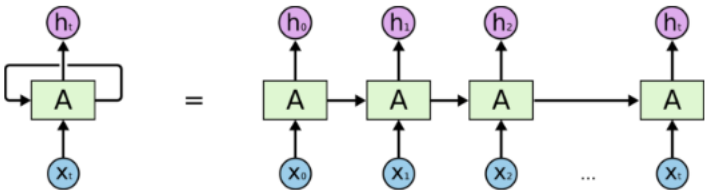
$$w_i = w_i - \alpha * \Delta l$$

4) עושים את שלב 3 לכל המשקולות והמודל שלנו למד! אפשר לעשות את שלב זה מספר פעמים, כי לפעמים יש יותר מדי מידע וצריך שהמודל ילמד אותו כמה פעמים. למספר הזה קוראים epochs.

RNN – Recurrent Neural Network

למדנו על מודל המורכב מנוירונים רבים שיכולים ללמוד דפוסים מורכבים מאוד. כעת, נמשיך למודל שצורתו דומה בקונספט אך שונה בהרכבתו ושימושו.

מודל ה-RNN (Recurrent Neural Network) הוא מודל שמאוד חזק בלנבא סדרות זמנים (timeseries). הוא בנוי כך שיש לו זיכרון פנימי, וכאשר מריצים את המודל על סדרת זמן הוא יכול לזכור טרנדים במידע ולהסיק מסכנות שרשת נוירונים רגילה לא יכולה להגיע אליהן.



An unrolled recurrent neural network.

בצורתו הכי בסיסית, למודל יש רק נוירון אחד בלבד ובו פונקציית activation. המודל הזה לוקח מידע שצורתו היא Timeseries (סדרת זמנים), לדוגמא דאטאסט של מזג האוויר לאורך השנה. אחרי שמוגדר למודל "צעדי זמן" (timesteps) שזו כמות הצעדים במידע שהמודל ייקח כדי לחשב את התוצאה שלו – אז בדוגמא שלנו עם מזג האוויר המודל ייקח את המזג האוויר ב-60 ימים האחרונים, המודל יריץ forward pass בצורה הבאה:

1) למודל קיים זיכרון פנימי שצבר ניסיון במהלך כל תהליך הלמידה. בצעד זמן הראשון, המודל מקבל את מזג האוויר ביום הראשון ומחשב פרדיקציה.
2) המודל מעביר את הפרדיקציה הזו לצעד זמן השני, שמקבל את אותה פרדיקציה ואת המזג האוויר ביום השני. המודל מחשב פרדיקציה חדשה עבור הצעד זמן השני.
3) המודל מעביר את הפרדיקציה הזו לצעד זמן השלישי, וכך הלאה עד שמגיעים לפרדיקציה אחרונה סופית.

האלגוריתם הזה הוא מה שהופך RNN למיוחד וחזק כשזה מגיע לפרדיקציה של מידע בצורה של סדרת זמן. אמנם המודל הבסיסי של RNN שראינו פה לא אופטימלי מכיוון שהוא יחסית חלש ולא יכול לזהות טרנדים מורכבים מאוד, העיקרון שלו הוא בסיס חזק למודלים מודרניים ומורכבים כמו GRU או LSTM שיכולים לזהות טרנדים מורכבים. כעת נלמד איך מלמדים RNN.

BPTT – Backpropagation Through Time

BPTT (Backpropagation Through Time) זה אלגוריתם למידה שהיא דומה מאוד לBackpropagation. העיקרון נשאר: ללמד את המודל בדרך רקורסיבית, כך שכל איטרציה המודל מקטין את השגיאה שלו. ב-RNN, הצורה של המודל שונה מרשת נוירונים רגילה, ולכן אלגוריתם הלמידה שלו צריכה להתאים לו. BPTT עובר על כל צעד זמן שה-forward pass עשה, מחשב לו ספציפית את השגיאה (בעזרת פונקציית השגיאה שהוגדרה למודל), ושומר את השגיאה הזאת. כאשר מסיימים את כל צעדי הזמן, עושים ממוצע לשגיאות ומאמנים את כל המשקולות לפי השגיאה הממוצעת.

נעשה דוגמא:

נגדיר מודל f(x) שהוא RNN. הhyperparameters של המודל שלנו הם ככה:

1) 3 צעדי זמן
2) פונקציית אקטיבציה מסוג Sigmoid.

כעת נעשה את הforward pass. u_1 הוא המשקולת שקשורה לפרדיקציה של הצעד זמן האחרון, וl_x הוא הקומבינציה הלינארית של אותו צעד זמן, וt_x הוא הצעד זמן x:

$$l_1 = x_1 * w_1 + 0 * u_1 \rightarrow f(t1) = \sigma(l_1)$$

$$l_2 = x_2 * w_1 + f(t1) * u_1 \rightarrow f(t2) = \sigma(l_2)$$

$$l_3 = x_3 * w_1 + f(t2) * u_1 \rightarrow f(t3) = \sigma(l_3)$$

כעט נחשב את השגיאה הממוצעת:

$$L_{avg} = \frac{1}{n}\sum_{i=0}^{n} y - f(t_i)$$

אחרי שחישבנו את הערכים האלו, נעשה *BPTT* , כאילו על צעד זמן אחד של המודל:

$$\frac{d}{df(t3)} L_{avg} = -1$$

$$\frac{d}{dl_3} f(t3) = \left(\sigma(l_3)\right)' = \sigma(l_3) * \left(1 - \sigma(l_3)\right)$$

$$\frac{d}{dw_3} l_3 = (x_3 * w_3 + f(t2) * u_3)' = x_3$$

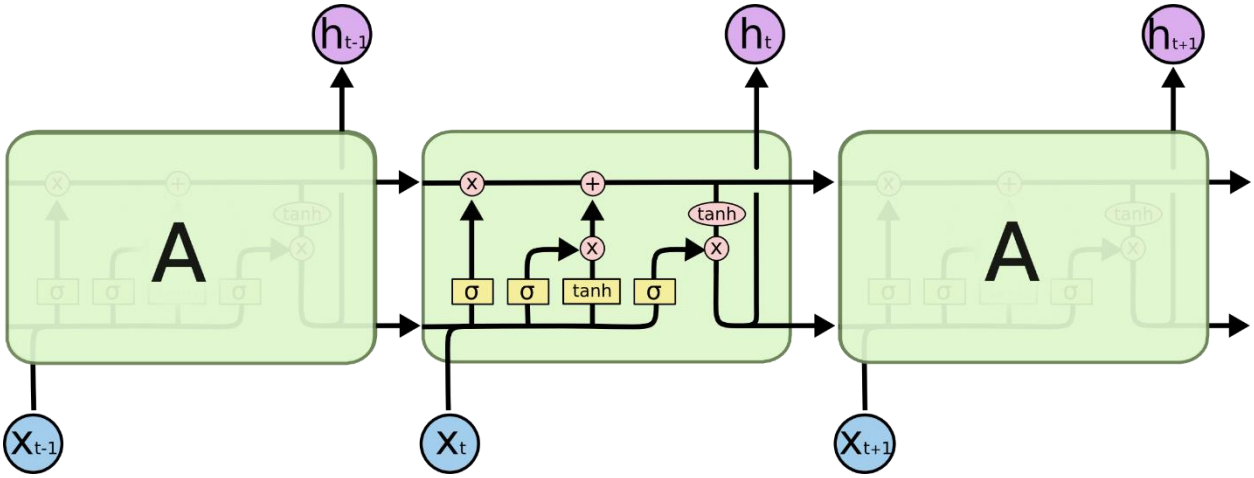
נשתמש בכלל השרשרת כמו בbackpropagation רגיל ונכפיל את שלושת הנגזרות:

$$w_3 = -1 * \sigma(l_3) * \left(1 - \sigma(l_3)\right) * x_3$$

וזהו! למדנו את המשקולת לפי מידע חדש. אפשר לראות שזה מאוד דומה לbackpropagation רגיל רק שהמודל לומד מעצמו וממידע חדש בו זמנית, ולכן היינו צריכים להתאים את אלגוריתם הלמידה אליו.

מודל ה-LSTM

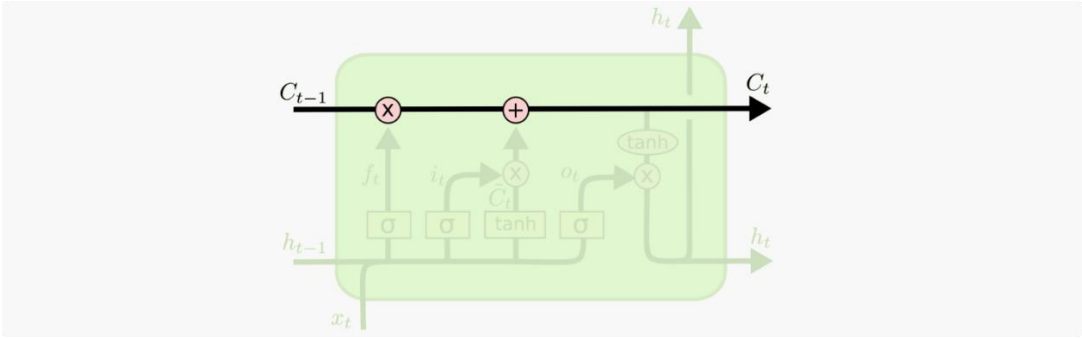
*כל התמונות מהחלק הזה לקוחות מהבלוג http://colah.github.io/posts/2015-08-Understanding-LSTMs/. בבקשה תקראו אותו, הוא מדהים!



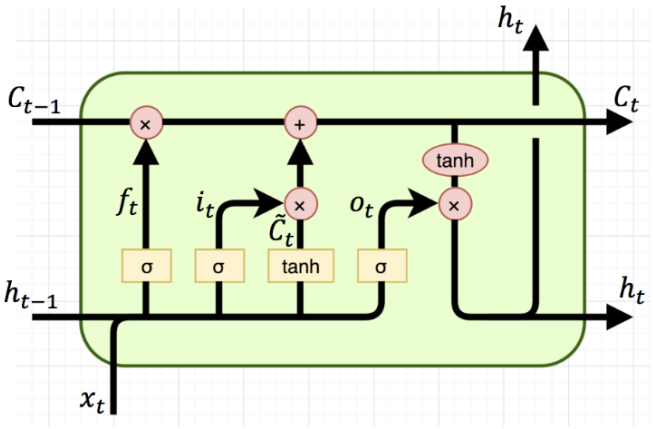
כדי לענות על שאלת המחקר: "האם אפשר לחזות מחירים של מניה? אם כן, מה אחוזי ההצלחה?" צריך להגדיר כמה תנאים:

1) המודל צריך להיות מסוגל לחזות כמה ימים בעתיד. נראה בהמשך שככל שמעלים את המספר של הימים כך אחוזי ההצלחה יורדים.
2) המידע שלנו הוא time series. כלומר, רצף זמנים שווה מתאריך אחד לתאריך אחר. המודל שלנו צריך להיות מסוגל להבין ולפרש את הדפוסים והטרנדים שיש במידע.

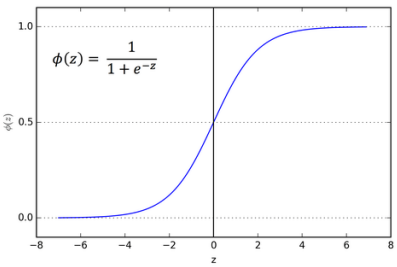
המודל שעונה לנו על התנאים האלה הוא מודל ה-LSTM. מודל ה-LSTM בנוי על רקע של מודל RNN (שלמדנו לפני), אך הוא שונה. בשונה מRNN הפשוט שלמדנו לפני, שיש לו נוירון אחד בלבד עם פונקציית activation, לנוירון (יותר מקובל לקנא אותו כ"תא") של LSTM יש זיכרון פנימי בנוסף שמטרתו לשמור מידע על טרנדים בtimeseries.



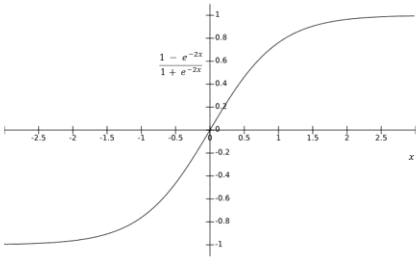
לפני שאסביר איך מודל LSTM עובד, אעבור על האותיות המתמטיות:



1) X_t: הקלט מן הדאטאסט בצעד זמן t.
2) h_t: הפלט של הצעד זמן הנוכחי, או במילים אחרות, הפרדיקציה של צעד זמן t. מובן מפה שh_{t-1} הוא הפלט של הצעד זמן t-1
3) C_t: הזיכרון הפנימי של הצעד זמן הנוכחי. מובן מפה שC_{t-1} הוא הזיכרון הפנימי של הצעד זמן t-1
4) $f_t, i_t, \tilde{C}_t, o_t$: אלה השערים. נגיע אליהם בהמשך.
5) σ: פונקציה הנקראת Sigmoid. הפונקציה מקבלת ערך X ומחזירה מספר בין 0 ל1:



6) $tanh$: פונקציה שמקבלת ערך X ומחזירה מספר בין (-1) ל1:



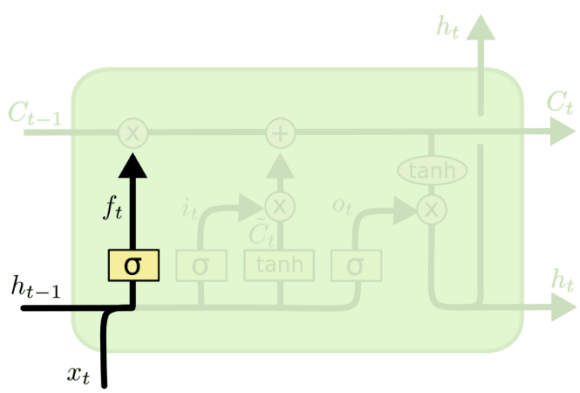
השערים

מי שאחראי על למצוא טרנדים ולקשר בין הזיכרון הפנימי לכל השאר הם פונקציות הנקראות "השערים". קיימים שלושה שערים: שער הקלט (input gate), שער השכיחה (forget gate), ושער הפלט (output gate). לכל שער תפקיד שונה.

התא פועל ב4 צעדים: חישוב שער השכיחה, חישוב שער הקלט, עדכון של הזיכרון הפנימי, וחישוב שער הפלט.

- כשיופיע $W_{gate} \cdot [h_{t-1}, x_t]$ הכוונה היא שW מחולק לשני משקולות, אחד לh_{t-1} ואחד לx_t. אפשר לחשוב על זה ככה: $W_{gate-h_{t-1}}$ מוכפל בh_{t-1} וW_{gate-x_t} מוכפל בx_t, והמשקולות ביחד מיוצגות על ידי W_{gate}. אם זה לא ברור מספיק, הסבר יופיע בחלק המעשי.

שער השכיחה



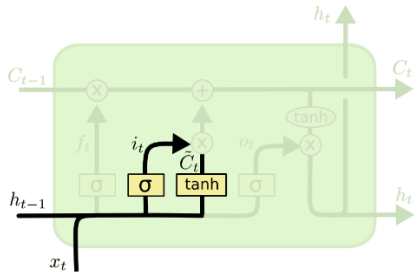
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

שער השכיחה תפקידו להעיף מידע לא רלוונטי מן החלק של הזיכרון. הוא עושה את זה כך:

1) עושים קומבינציה לינארית של הדאטא הנוכחי והפלט של הצעד זמן הקודם עם המשקולות של שער השכיחה($W_f \cdot [h_{t-1}, x_t]$), ומוסיפים b_f. זה מייצג פרדיקציה יחסית פרימיטיבית, כמו שראינו בRNN רגיל.

2) את הפרדיקציה הזאת נכניס לתוך פונקציית הSigmoid, ונקבל ערך בין 0-1 שמייצג עד כמה הפרדיקציה הזאת הצליחה (אפשר לחשוב עליו כמו אחוז בין 0% ל100% שמתאר לנו כמה בטוח המודל שלנו בתשובה שלו). לזה נקרא f_t

שער הקלט



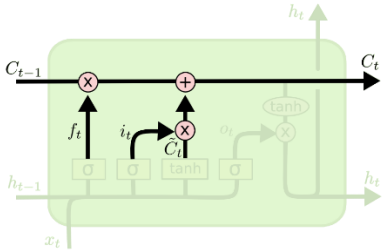
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

שער הקלט מטרתו להוסיף מידע חדש לזיכרון הפנימי. הוא עושה את זה כך:

1) עושים את אותו תהליך ששער השכיחה עשה, רק אם המשקולות של שער הקלט W_i. מכניסים את הפרדיקציה הזאת לתוך פונקציית Sigmoid. לזה נקרא i_t
2) עושים קומבינציה לינארית של הדאטא הנוכחי והפלט של הצעד זמן הקודם עם המשקולות האחרות של שער הקלט($W_c \cdot [h_{t-1}, x_t]$), ומוסיפים b_f. זה מייצג פרדיקציה יחסית פרימיטיבית, כמו שראינו בRNN רגיל. מכניסים את הפרדיקציה הזאת לפונקציית $tanh$. לזה נקרא $\tilde{C}_t$

עדכון הזיכרון הפנימי

אחרי שמחשבים את הערכים של שער השכיחה ושער הקלט, נרצה לעדכן את הזיכרון הפנימי עם הערכים שחישבנו.



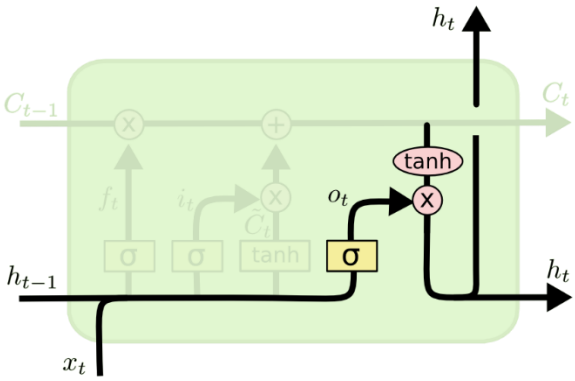
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

החלק הזה יחסית פשוט:

1) ניקח את הערך f_t שקיבלנו משער השכיחה ונכפיל אותו עם הזיכרון הפנימי C_{t-1}. מה שקורה כאן זה שהמודל, לפי שער השכיחה, מחליש את המידע שיושב לו בזיכרון הפנימי. אפשר לחשוב על זה כאילו המודל מקבל מידע חדש ועל סמך מידע זה מחליט להעיף זיכרונות שיכולות לפגוע בפרדיקציות עתידיות.
2) מוסיפים לזיכרון הפנימי את המידע החדש. המידע הזה הוא הכפלה של i_t ו-$\tilde{C}_t$.

כעת הזיכרון הפנימי מעודכן לצעד זמן הנוכחי. נשאר לנו רק חישוב שער הפלט והוצאת הפלט.

שער הפלט



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

שער הפלט אחראי על חישוב הפרדיקציה הסופית:

1) עושים את אותו תהליך ששער השכיחה וקלט עשו, רק אם המשקולות של שער הפלט W_o. מכניסים את הפרדיקציה הזאת לתוך פונקציית Sigmoid. לזה נקרא o_t
2) מכניסים את הזיכרון הפנימי לתוך פונקציית $tanh$.
3) מכפילים את התוצאות של 1) ו-2), ומקבלים את הפרדיקציה של אותו צעד זמן h_t.

המודל כעט מעביר את הזיכרון הפנימי C_t והפרדיקציה h_t לצעד זמן הבא, וכך אלה עד שמקבלים פרדיקציה סופית.

ה-forward propagation

כמו לרשת נוירונים רגילה, גם לLSTM קיים forward propagation. אמנם בRNN ראינו שזה רק כולל חישוב אחד של פרדיקציה, ושLSTM בנוי על רקע RNN, למודל LSTM יש forward propagation יותר מורכב. כעת נעבור על האלגוריתם:

לפי כמות הצעדים בtimeseries (שזה פשוט האורך שלו), בכל צעד זמן:

1) נחשב את הערך של שער השכיחה.
2) נחשב את הערך של שער הקלט
3) נעדכן את הזיכרון הפנימי
4) נחשב את הפלט בעזרת שער הפלט
5) נעביר את הפלט ואת הזיכרון הפנימי לצעד זמן הבא

בסוף נקבל פלט סופי וזה תהיה הפרדיקציה הסופית של המודל.

כפי שראינו, למודל ה-LSTM יש את היכולת לזכור דפוסים במידע שהוא מקבל וזה משפיע על הפרדיקציות שלו, ובנוסף לכך אנחנו יכולים לתת למודל את הפרדיקציות של עצמו כדי לנבא מספר ימים נוספים בעתיד. לכן זה עונה על התנאים שהצבנו בשביל מודל כדי לנבא מחירי מניות.

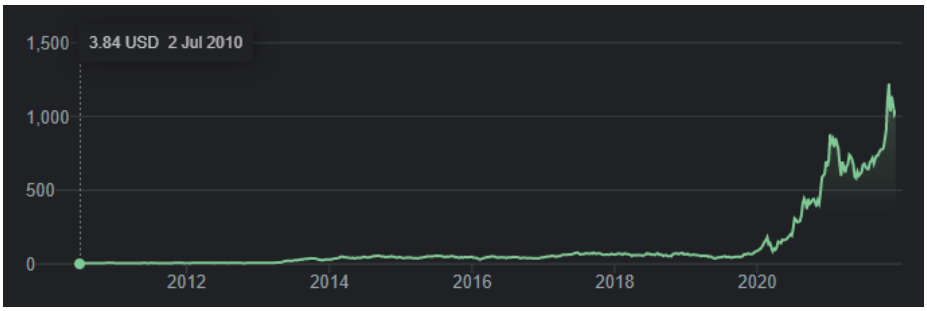
הדאטא

הפעולה של למידת מכונה לא יכולה להתקיים ללא דאטא. בפרויקט של למידת מכונה, תהליך אסיפה וה-QA של הדאטא הוא החלק הכי חשוב בפרויקט. בלי דאטא נקי ו-"לא משוחד" (unbiased), המודל שלנו ילמד דפוסים שאנחנו לא מחפשים. לכן חייבים לשים לב לקונטקסט של הדאטא שלנו.

התהליך מתחיל בכך שנשאל את עצמנו מה המטרה של המודל שלנו. במקרה שלנו אנחנו רוצים שהוא יקבל סדרת זמן של מחיר המניה, ויחשב לנו פרדיקציה של שווי המניה לכמה ימים בעתיד (לפי החלטתנו). כרגע אנחנו יכולים להגיד בוודאות שהדאטא הכי בסיסית שאנחנו צריכים לפרויקט שלנו הוא מחיר המניה.

שנית, אנחנו צריכים להחליט איזו מניה לעשות אליה פרדיקציה. מכיוון ששאלת החקר שלי היא "האם אפשר לחזות מחירי מניה? אם כן מה אחוזי ההצלחה" החלטתי לבחור בשתי מניות כדי לבדוק את ההצלחה של המודל:

1) מניית TSLA. למניה יש היסטוריה של 2500 ימים. אפשר לראות בתמונה למטה של-2000 ימים קצב השינוי במחיר של המניה אחידה, וב-500 ימים האחרונים הקצב שלה היא כמעת בלתי צפויה. הסיבה לכך היא מורכבת מאוד ולא כל כך רלוונטית, אבל חשוב להכיר בהתנהגות הזאת. מכיוון שמניה זו היא קיצונית בקצב שינויה היא תשמש כדוגמא טובה לאייך המודל שלנו יגיב אחרי שלמד סדרת זמן אחידה ופתאום נתקל בטרנד שלא למד.



2) S&P500 Index: מניה זו היא מאוד רגועה ותמיד הייתה אחידה בקצב שינויה. לכן היא תהיה דרך טובה למדוד איך המודל שלנו מגיב לסדרת זמן רגועה.

באתר של Yahoo! Finance אפשר להוריד קבצי csv. לשני המניות הללו. הורדתי שתי קבצים:

1) TSLA3.csv – קובץ באורך של 2510 שורות שיש בו את מחיר המניה TSLA מהתאריך 26-07-2011 ועד 26-07-2021. סך הכול 10 שנים של דאטא על המניה.
2) SNP500.csv – קובץ באורך של 2473 שורות שיש בו את מחיר המניה SNP500. התאריכים לא ידועים לי אבל ההיקף הוא 10 שנים גם כן.

כעט צריך להכין פונקציות בקוד שיעשו import לקבצים הללו. נכתוב:

```
def get_data(data_length) -> pd.DataFrame:
    # Import the CSV's ('Path', 'Delimiter', [Columns to use])
    paths_info = [('TSLA3.csv', ',', [3]), ]
    datasets = [np.genfromtxt(f"Data\{path[0]}", delimiter=path[1], usecols=path[2]) for path in paths_info]

    # Add Bias
    datasets = np.insert(datasets, len(datasets), np.zeros(data_length,), axis=0)
    data = np.stack(datasets, axis=-1)

    return data
```

הפונקציה get_data מקבלת את האורך של סדרת הזמן/הדאטא. זאת מכיוון שאם יש לנו יותר מ-feature אחד אנחנו נרצה שהקלט שנביא למודל יהיה אחיד באורכו.

השורה הראשונה שומרת מידע אל כל קובץ csv בתוך רשימה. המידע המדויק הוא: path, delimiter, columns to use. לדוגמא בקובץ SNP500.csv מחיר המניה נמצאת בטור שלוש ולכן צריך להגיד את זה.

השורה השנייה משתמשת בlist comprehension כדי לעבור על כל קובץ שהכנסנו לpaths_info ועושה לקובץ import בעזרת הפונקציה numpy.genfromtxt. את המידע היא שומרת במערך.

השורה השלישית מכניסה שורה של אפסים בסוף המערך בשביל הbias. זוכרים מה זה? הסיבה שאני מכניס אותו פה הוא בשביל גמישות. אם ארצה לשנות את הערך ההתחלה מאפס לערך אחר, אני רק צריך לשנות את זה פה ולא במלא מקומות במודל. דוגמא:

```
[[  5.6        5.528      5.634    ... 625.219971 623.900024 605.119995]
 [  0.         0.         0.       ...   0.         0.         0.      ]]
```

השורה הרביעית והאחרונה עושה פעולה של stack על המערך שהחזיק את הדאטאסטס. במקום שהיה נראה כמו התמונה האחרונה (כל דאטאסט הוא בשורה נפרדת, וכל טור הוא צעד זמן שונה), הוא נראה כך (כל דאטאסט הוא בטור נפרדת, וכל שורה הוא צעד זמן שונה):

```
[[  5.6        0.      ]
 [  5.528      0.      ]
 [  5.634      0.      ]
 ...
 [625.219971   0.      ]
 [623.900024   0.      ]
 [605.119995   0.      ]]
```

זה צורה שהרבה יותר נוחה לעבוד איתה.

אחרי שיש לנו את הדאטאסט הסופי, צריכים לחלק אותו לtraining set ו-test set. נהוג לחלק ביחס של 20%-80%, 80% לtraining ו-20% לtest. נכתוב פונקציה שתעשה בדיוק זה, אך שאפשר להגיד ספציפית מה הכמות לכל set כדי שיהיה אפשר לשנות בעתיד:

```python
def split_train_test(data, timesteps, length, train_length) -> tuple:
    training = []
    testing = []
    for i in range(timesteps, train_length):
        training.append([data[i-timesteps:i], data[i, 0]])

    for i in range(train_length, length):
        testing.append([data[i-timesteps:i], data[i, 0]])

    return np.array(training, dtype=object), np.array(testing, dtype=object)
```

הפונקציה מקבלת את הדאטאסט, אורך timesteps, אורך length המסמל את האורך של כל הדאטא ואורך train_length המסמל את האורך שנרצה לתת לtraining set. אורך הtesting set יהיה:

$$testSetLen = length - train\_length$$

הפונקציה עוברת מ-timesteps ועד לtrain_length כדי להכין מערך שבתוכו יש סדרות זמנים קטנים באורך timesteps. אלו יהיו הדגימות מידע שנתן למודל בכל פעם.

הפונקציה מחזירה את שני הsets, ראשון הtraining ושני הtesting.

בחירת פרמטרים למודל

עכשיו שיש לנו את הפונקציות שמושכות ומכינות את הדאטא, ומודל LSTM מוכן, אפשר להתחיל את שלב הניסויים. המטרה של שלב זה הוא לנסות למצוא את הפרמטרים האופטימליים למודל שלנו כמו:

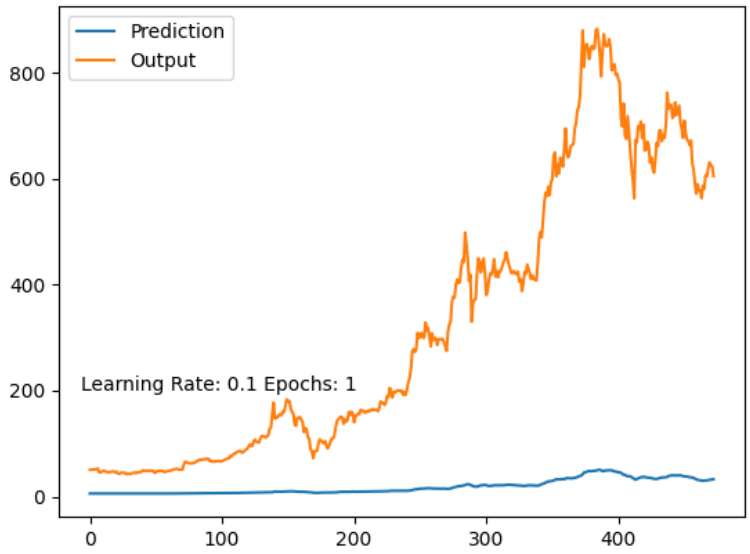
Learning rate, Cell Count, Timesteps.

נתחיל אם הערכים האלו:

Learning Rate	0.1
Cell Count	1
Timesteps	50

אלו ערכים שבחרתי משרירות, כלומר אין מאחוריהם סיבה לבחירתם חוץ מזה שהם נראים בסיסיים.

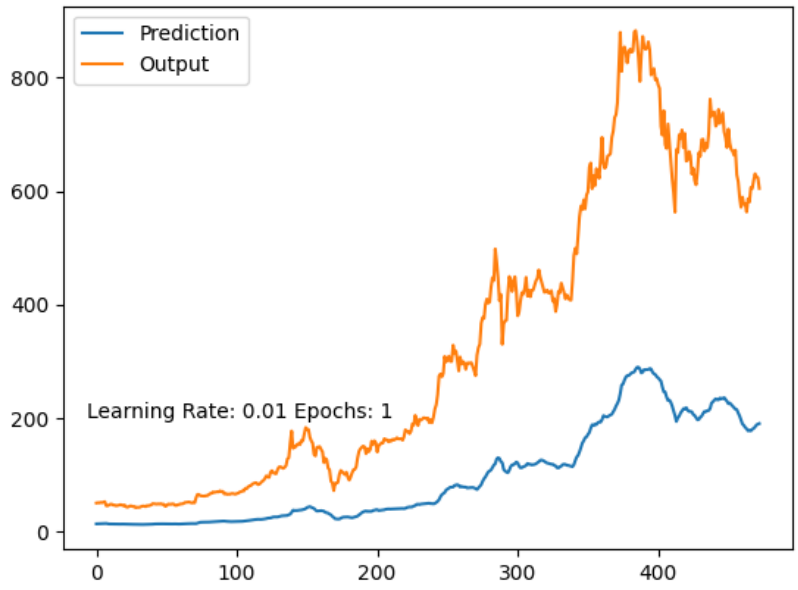
נריץ את המודל שלנו:



- זמן הריצה הייתה 11.7 שניות, השגיאה היא 306.9

אפשר לראות שהמודל שלנו פספס בגדול. לפני שנשנה פרמטר, ננסה להבין למה זה קרה. אם נתבונן בשני הגרפים נראה שהמודל שלנו לא מצליח ללמוד את הדפוס ברמה המספרית, כלומר הוא כן עוקב אחרי הטרנד אבל הוא "מכווץ". מפה נובע שצריך להוריד את הlearning rate. נריץ סבב שני:

Learning Rate	0.01
Cell Count	1
Timesteps	50



- זמן ריצה הייתה 11.2 שניות, השגיאה היא 229.1

קיבלנו תוצאה מדהימה! אפשר לראות שהפרדיקציה שלנו עשתה Scale יותר טוב, וזה ממש מתחיל להיראות דומה לדפוס המקורי. אני אריץ את המודל עם learning rates שונים ואשים אותם בטבלה עם השגיאה של אותו ריצה (מכיוון שפרדיקציה היא לא תמיד אותו הדבר, נריץ את המודל 4 פעמים):

291 Average over 10 runs	0.001
234.5 Average over 10 runs	0.0001
243.5 Average over 10 runs	0.00001

אפשר לראות ש0.0001 ייתן לנו את השגיאה המינימלית, ולכן נלך אתו.

עכשיו נתחיל לעלות את ה-Cell Count. ככל שיש יותר תאים, המודל ילמד יותר דפוסים. אריץ את המודל עם כמות תאים שונים מספר פעמים ואשים את התוצאות בטבלה:

2 Cells	253 Average over 10 runs
4 Cells	185.6 Average over 10 runs
6 Cells	195.6 Average over 10 runs
8 Cells	100.6 Average over 10 runs
10 Cells	155.2 Average over 10 runs

בירור לנו פה ש-8 תאים הוא הכמות האופטימלית. רק כדי לבדוק, הרצתי את המודל 10 פעם עם 9 תאים מכיוון שזה בין השני ערכים הנמוכים. התוצאה:

9 Cells	169.8 Average over 10 runs

כעט נמשיך לבדוק את הפרמטר הבא אם 8 תאים, שזה הצעדי זמן (timesteps):

50	174 Average over 10 runs
100	166.8 Average over 10 runs
150	132.6 Average over 10 runs
200	170 Average over 10 runs
250	188 Average over 10 runs
300	123.4 Average over 10 runs
350	152.4 Average over 10 runs

אפשר לראות שיש פה חוסר עקביות בתוצאות שלנו. לדוגמא, פעם אחת עם 50 צעדי זמן ו-8 תאים קיבלנו ממוצת שגיאה של 100, ובדיוק אחרי זה קיבלנו ממוצת שגיאה של 174 עם אותה כמות צעדי זמן ותאים. לכן, נשנה את הגישה שלנו.

כדי לבדוק מה הפרמטרים הכי טובים למודל שלנו, נריץ אלגוריתם אופטימיזציה שנקראת grid search. אלגוריתם זה הוא מאוד פשוט, והדבר היחיד שעושים זה מנסים את כל הקומבינציות האפשריות. מכיוון

שהדאטא שיש לנו הוא כמות מאוד קטנה (יחסית), ומטרת המודל שלנו הוא לימודית ולא להפקה אנחנו יכולים להרשות לעצמנו להשתמש באלגוריתם זה.

האלגוריתם שלנו יעבור על כל הקומבינציות בטבלה שלמטה. מכיוון שבינה מלאכותית זה דבר רנדומלי, אנחנו יכולים לקבל תשובות שונות בכל פעם שנריץ קומבינציה. לכן נריץ כל קומבינציה 20 פעם ונעשה ממוצע של ההפסד. הטבלות:

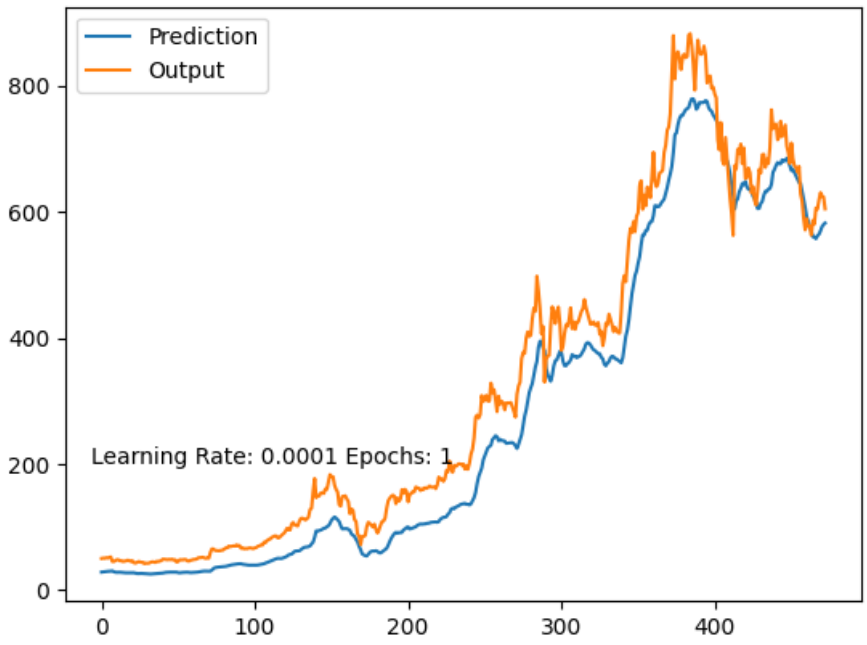
על מניית TSLA:

	Steps: 50	Steps: 100	Steps: 150	Steps: 200	Steps: 250	Steps: 300	Steps: 350
Cells: 2	245	241	259	245	227	221	260
Cells: 4	221	209	210	217	216	225	202
Cells: 6	200	193	198	185	193	184	182
Cells: 8	171	153	171	170	154	168	174
Cells: 10	150	155	153	124	161	148	143

על מניית S&P500:

	Steps: 50	Steps: 100	Steps: 150	Steps: 200	Steps: 250	Steps: 300	Steps: 350
Cells: 2	1585	1791	1769	1731	1805	1743	1717
Cells: 4	1485	1474	1416	1407	1433	1492	1479
Cells: 6	1377	1347	1443	1294	1422	1270	1379
Cells: 8	1307	1226	1162	1222	1161	1178	1130
Cells: 10	882	926	1062	812	809	983	986

אפשר לראות שבשני המקרים 10 תאים ו200 צעדי זמן מביא לנו את התוצאות הכי טובות. לכן, נשתמש בפרמטרים אלו. לדוגמא, הנה פרדיקציה על המנייה של TESLA:



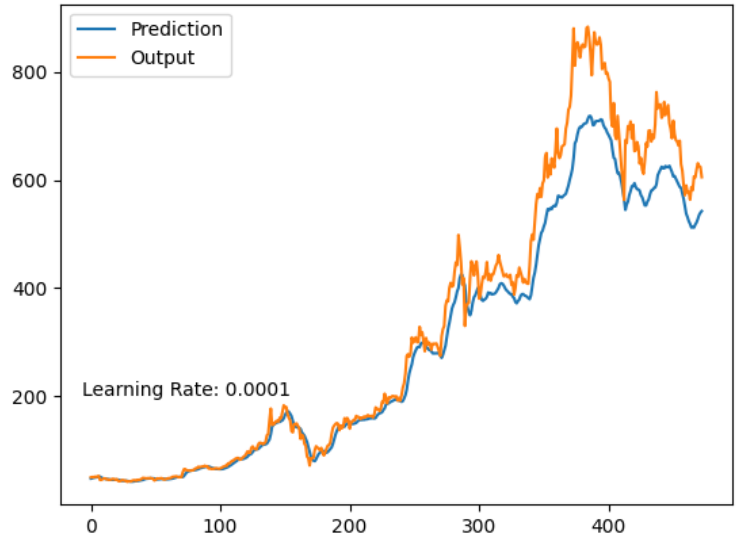
זהו תוצאה מאוד מפתיעה למודל מאוד בסיסי, ללא שיפורים מקצועיים. זה רק מראה את הכוח של למידה עמוקה.

השוואת המודל שלנו עם מודל מקצועי

כעט נכין מודל בעזרת ספרייה מוכרת (Keras) כדי לראות את ההבדל בין המודל שלי למודל שמקצוענים כתבו. נשתמש באותו סוג מודל, אותם פרמטרים, ואותו דאטאסט:

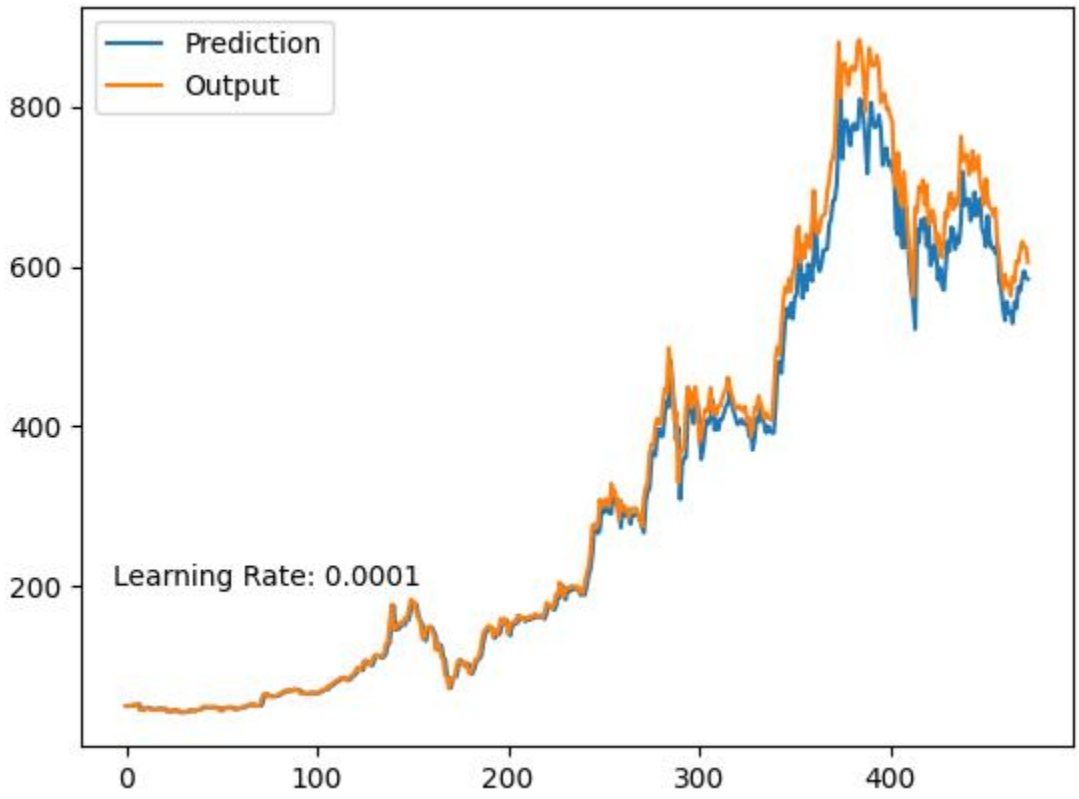
```
x_train = []; y_train = []
for sample in training:
    x_train.append(sample[0]); y_train.append(sample[1])

x_test = []; y_test = []
for sample in testing:
    x_test.append(sample[0]); y_test.append(sample[1])

x_train = np.array(x_train); y_train = np.array(y_train)
x_test = np.array(x_test); y_test = np.array(y_test)


model = keras.models.Sequential()
model.add(keras.layers.LSTM(units=cell_count, input_shape=(x_train.shape[1], x_train.shape[2])))
model.add(keras.layers.Dense(units = 1))

model.compile(loss = 'mean_squared_error', optimizer=keras.optimizers.RMSprop(learning_rate=learning_rate))
model.fit(x_train, y_train, epochs = epochs, batch_size = 32)
```

הנה התוצאה:



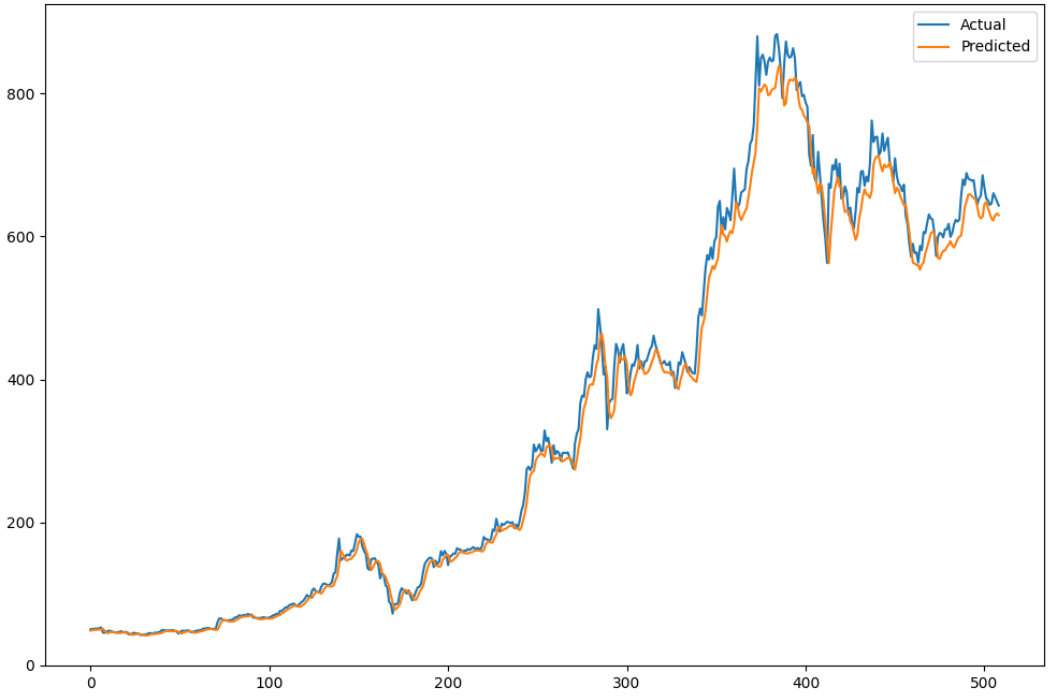
אפשר לראות שהמודל הרבה יותר מדויק בהתחלה, אבל בהמשך מגיע לתוצאות דומות למודל שלנו.

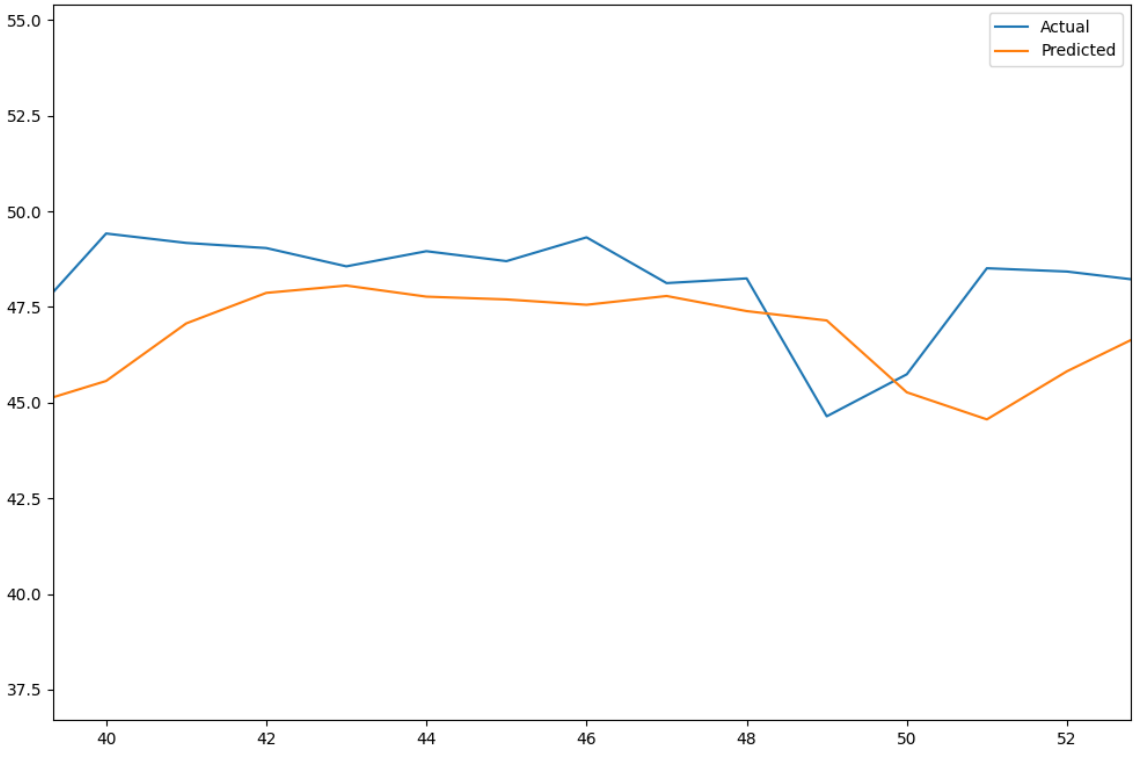
מכיוון שזה מעניין, נבדוק את התוצאות של אותו מודל בדיוק, רק עם שימוש בADAM, סוג של פונקציית למידה מתקדמת וחדשה:



התוצאות מאוד מרשימים. המודל הצליח לחזות את הדפוס כמעט מדויק עד היום ה-300. משם, יש מאוד קצת סטייה. איזה מדהים!

אבל האם באמת אפשר לחזות מחירי מניות? התשובה היא לא. אם נשים לב, ונעשה זום לראות איזו פרדיקציות המודל עושה כל יום נראה שהוא רק חצי מהזמן מצליח לחזות נכון עלייה וירידה. זה הופך את הפרדיקציות ל-לא אמינות ולכן אי אפשר באמת להשתמש במודל של AI כדי לעשות Day Trading ולהרוויח כסף.


Actual
Predicted
800
600
400
200
0
0
100
200
300
400
500


Actual
Predicted
55.0
52.5
50.0
47.5
45.0
42.5
40.0
37.5
40
42
44
46
48
50
52

אפשר לראות שהמודל מתעכב בפרדיקציות שלו, ואם לא מפספס לגמרי – אם היית ביום 47 רואה שהמנייה תרד, היית מוכר. ביום 49 המנייה באמת ירדה, ואתה מסתכל פרדיקציה של המודל ורואה שביום 50 המנייה תמשיך לרדת. אם הקשבת למודל ומכרת, הייתה מפסיד כסף מכיוון שבמציאות ביום 50 המנייה עלתה. זה קורה גם בימים היותר אי וודאים:

אם היית ביום 352 ורואה שהמודל מנבא העלה במחיר וקונה 5 מניות, היית מפסיד בערך 200 דולר!

מסקנה וסיכום

הגענו למסקנה ששיערנו בתחילת העבודה: אי אפשר לחזות מחירי מניות ברמה היום יומית. אולי אם הייתי עם הרבה שנים של ניסיון בAI ו Data Science הייתי יכול לקבל מסכנה יותר מורכבת ומדויקת, אבל המטרה של עבודה זו הייתה ללמוד בינה מלאכותית ולמידה של דפוסי זמן, ולדעתי הצלחתי לעשות זו.

אני מקווה שמי שקורא את עבודה זה גם כן למד, ויכול להבין קצת יותר לעומק שמדברים על בינה מלאכותית. אם לא העברתי את החומר במובן, אני מצטער על בזבוז הזמן. אבל בכל זאת תודה שקראת עד לפה, מכיוון שעבדתי מאד קשה על העבודה הזאת ואני מאד גאה בה.

הרבה תודות לזוהר ברונפמן, המנחה שלי, שעזר לי במהלך העבודה ובלבחור את הנושא מההתחלה. לא יכולתי להגיע לשלב זה בלעדיו.

הקוד

המודל שלי

```python
from datetime import time
from numpy.random.mtrand import uniform
import pandas as pd
import numpy as np
from numpy.random import uniform
from sklearn.preprocessing import MinMaxScaler
import math
import matplotlib.pyplot as plt
from datetime import datetime
import time


class LSTMCell:
    def __init__(self, features):
        self.hidden = 0
        self.previous_output = 0

        # Weights. Each row represents a different gate, ex: 0 -> forget, 1 -> input, etc...
        self.hdn_w = np.random.rand(4, features) * math.sqrt(1/(features + 1))
        self.inp_w = np.random.rand(4, features) * math.sqrt(1/(features + 1))


    @staticmethod
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    def forget_gate(self, x, prev_y) -> tuple:
        linc = sum(x * self.inp_w[0] + prev_y * self.hdn_w[0])
        gate_output = self.sigmoid(linc)
        return gate_output, linc

    def input_gate(self, x, prev_y):
        linc = sum(x * self.inp_w[1] + prev_y * self.hdn_w[1])
        gate_output = self.sigmoid(linc)

        linc2 = sum(x * self.inp_w[2] + prev_y * self.hdn_w[2])
        return gate_output, linc, linc2

    def output_gate(self, x, prev_y):
        linc = sum(x * self.inp_w[3] + prev_y * self.hdn_w[3])
        gate_output = self.sigmoid(linc)
        return gate_output, linc

    def predict(self, x):
        for time_step in range(len(x)):

            # Forget Gate
            forget_gate, _ = self.forget_gate(x[time_step],
                                              self.previous_output)


            # Input Gate
            input_gate, lci1, lci2 = self.input_gate(x[time_step],
                                                     self.previous_output)

            self.hidden *= forget_gate
            self.hidden += input_gate * np.tanh(lci2)
```

```python

            # Output Gate
            output_gate, _ = self.output_gate(x[time_step],
                                              self.previous_output)

            self.previous_output = np.tanh(self.hidden) * output_gate

        return self.previous_output


class Model:
    def __init__(self, features, cell_count):
        self.features = features
        self.cell_count = cell_count

        # Create the cells & weights and initalize them
        self.cells = [LSTMCell(self.features) for _ in range(self.cell_count)]
        self.weights = np.random.rand(cell_count) * math.sqrt(1/(cell_count))

    @staticmethod
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return self.sigmoid(x) * (1 - self.sigmoid(x))

    def predict(self, x):
        # Pass through our input to each cell and collect the predictions
        cell_predictions = []
        for cell in self.cells:
            cell_predictions.append(cell.predict(x))

        # Combine the predictions with the model weights
        linear_combination = sum(np.array(cell_predictions).T * self.weights)
        return linear_combination

    def fit(self, x, y, epochs, learning_rate):
        """
        Using the input information and cell functions, predicts a value 1 timestep in the future by
        fitting the input dataset on the cell using backpropagation through time (BPTT)
        :param x: Numpy Array of size (timesteps, features) representing the input
        :return: A number representing a prediction
        """

        for epoch in range(epochs):
            #print("Starting Epoch " + str(epoch))
            time_start = time.time()

            for batch in x:

                hidden_states = [[] for _ in range(self.cell_count)]
                linc_f = [[] for _ in range(self.cell_count)]
                linc_i1 = [[] for _ in range(self.cell_count)]
                linc_i2 = [[] for _ in range(self.cell_count)]
                linc_o = [[] for _ in range(self.cell_count)]
                input_gates = [[] for _ in range(self.cell_count)]
```

```python
                hidden_before_forget = [[] for _ in range(self.cell_count)]
                cell_outputs = [[] for _ in range(self.cell_count)]
                overall_outputs = []
                linc_overall = []

                # Forward Pass
                for time_step in range(len(batch)):

                    for num, cell in enumerate(self.cells):

                        # Forget Gate
                        forget_gate, lcf = cell.forget_gate(batch[time_step], cell.previous_output)
                        cell.hidden *= forget_gate

                        hidden_before_forget[num].append(cell.hidden)

                        # Input Gate
                        input_gate, lci1, lci2 = cell.input_gate(batch[time_step], cell.previous_output)
                        cell.hidden += input_gate * np.tanh(lci2)

                        # Output Gate
                        output_gate, lco = cell.output_gate(batch[time_step], cell.previous_output)
                        output = np.tanh(cell.hidden) * output_gate
                        cell.previous_output = output
                        cell_outputs[num].append(output)

                        linc_f[num].append(lcf), linc_i1[num].append(lci1), linc_i2[num].append(lci2), linc_o[num].append(lco)
                        input_gates[num].append(input_gate)
                        hidden_states[num].append(cell.hidden)

                    # Combine each cell output into a linear combination and sigmoid it
                    linc_output = sum([cell_output[-1] * self.weights[num] for num,cell_output in enumerate(cell_outputs)])
                    linc_overall.append(linc_output)
                    overall_outputs.append(self.sigmoid(linc_output))

                # Back propagate for each time step
                hdn_gradients = np.zeros(shape=(self.cell_count, 4, self.features))
                inp_gradients = np.zeros(shape=(self.cell_count, 4, self.features))
                model_gradients = np.zeros(shape=self.cell_count)

                for time_step in range(1, len(batch)+1):
                    overall_outputs = np.array(overall_outputs)

                    # Updating Model Weights
                    d_model = (overall_outputs[-time_step] - y[-time_step]) * (self.sigmoid_derivative(linc_overall[-time_step]))
                    model_gradients += d_model * np.array([z[-time_step] for z in cell_outputs])

                    for num,cell in enumerate(self.cells):
                        # Updating Output Weights
                        d_output = d_model * (self.weights[num]) * (np.tanh(hidden_states[num][-time_step])) * (
                                self.sigmoid_derivative(linc_o[num][-time_step]))
```

```
                        hdn_gradients[num] += d_output * hidden_states[num][-time_step]
                        inp_gradients[num] += d_output * batch[-time_step]


                        # Updating Input2 Weights
                        d_input2 = d_model * (self.weights[num]) * (cell_outputs[num][-time_step]) * (
                                    1 - np.power(np.tanh(hidden_states[num][-time_step]), 2)) * (
                                    self.sigmoid_derivative(hidden_states[num][-time_step])) * (input_gates[num][-time_step]) * (
                                    1 - np.power(np.tanh(linc_i2[num][-time_step]), 2))

                        hdn_gradients[num] += d_input2 * cell_outputs[num][-time_step]
                        inp_gradients[num] += d_input2 * batch[-time_step]

                        # Updating Input1 Weights
                        d_input1 = d_model * (self.weights[num]) * (cell_outputs[num][-time_step]) * (
                                    1 - np.power(np.tanh(hidden_states[num][-time_step]), 2)) * (
                                    self.sigmoid_derivative(hidden_states[num][-time_step])) * (np.tanh(linc_i2[num][-time_step])) * (
                                    self.sigmoid_derivative(linc_i1[num][-time_step]))

                        hdn_gradients[num] += d_input1 * cell_outputs[num][-time_step]
                        inp_gradients[num] += d_input1 * batch[-time_step]

                        # Updating Forget Weights
                        d_forget = d_model * (self.weights[num]) * (cell_outputs[num][-time_step]) * (
                                    1 - np.power(np.tanh(hidden_states[num][-time_step]), 2)) * (
                                    self.sigmoid_derivative(hidden_states[num][-time_step])) * (hidden_before_forget[num][-time_step]) * (
                                    self.sigmoid_derivative(linc_f[num][-time_step]))

                    hdn_gradients[num] += d_forget * cell_outputs[num][-time_step]
                    inp_gradients[num] += d_forget * batch[-time_step]

            self.weights -= learning_rate * (model_gradients / len(batch))
            for num, cell in enumerate(self.cells):
                cell.hdn_w -= learning_rate * (hdn_gradients[num] / len(batch))
                cell.inp_w -= learning_rate * (hdn_gradients[num] / len(batch))

            #print(f"- Time took: {time.time() - time_start}")


def get_data(data_length) -> pd.DataFrame:
    # Import the CSV's ('Path', 'Delimiter', [Columns to use])
    paths_info = [('TSLA3.csv', ',', [4]), ]
    datasets = [np.genfromtxt(f"Data\{path[0]}", delimiter=path[1], usecols=path[2], max_rows=data_length) for path in paths_info]

    # Add Bias
    datasets = np.insert(datasets, len(datasets), np.zeros(data_length,), axis=0)
    data = np.stack(datasets, axis=-1)

    return data
```

```python


def split_train_test(data, timesteps, length, train_length) -> tuple:
    training = []
    testing = []
    for i in range(timesteps, train_length):
        training.append([data[i-timesteps:i], data[i, 0]])

    for i in range(train_length, length):
        testing.append([data[i-timesteps:i], data[i, 0]])

    return np.array(training, dtype=object), np.array(testing, dtype=object)


cell_count = 10
for timesteps in [200]:

    # Prepare Data
    price_scaler = MinMaxScaler(feature_range=(0, 1))
    data = get_data(data_length=2473)


    data[:, 0] = price_scaler.fit_transform(np.reshape(data[:, 0], (-1, 1)))[:, 0]


    # Training, Validation, & Test Sets
    features = 2
    #timesteps = timesteps
    learning_rate = 0.0001
    epochs = 10
    #cell_count = cell_count
    training, testing = split_train_test(data, timesteps=timesteps, length=len(data), train_length=2000)

    # Create model and predict
    time_start = time.time()

    model = Model(features=features, cell_count=cell_count)
    model.fit(training[:, 0], training[:, 1], learning_rate=learning_rate, epochs=epochs)

    print(f"Time Taken: {time.time()-time_start}s")

    predictions = []
    real = []
    error = 0
    for x, y in zip(testing[:, 0], testing[:, 1]):
        prediction = model.predict(x)
        scaled = price_scaler.inverse_transform(prediction.reshape(1, -1))[0]
        scaled_real = price_scaler.inverse_transform(y.reshape(1, -1))[0]
        predictions.append(scaled)
        real.append(scaled_real)
        error += price_scaler.inverse_transform(y.reshape(1, -1))[0] - scaled


    plt.plot(predictions)
    plt.plot(real)
    plt.legend(["Prediction", "Output"])
```

```
    plt.text(s="Learning Rate: " + str(learning_rate) + " Epochs: " + str(epochs), x=-7.2, y=200)

    plt.show()
```

המודל של Keras

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dropout, Dense
from tensorflow.keras.metrics import MeanSquaredError
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt


def get_data(data_length):
    # Import the CSV's ('Path', 'Delimiter', [Columns to use])
    paths_info = [('SNP500.csv', ',', [3]), ]
    datasets = [np.flip(np.genfromtxt(f"Data\{path[0]}", delimiter=path[1], usecols=path[2]))
for path in paths_info]

    # Add Bias
    # datasets = np.insert(datasets, len(datasets), np.zeros(data_length,), axis=0)
    data = np.stack(datasets, axis=-1)

    return data


def split_train_test(data, timesteps, length, train_length, days_to_predict) -> tuple:
    x_train, y_train, x_test, y_test = [], [], [], []
    for day in range(timesteps, train_length-days_to_predict):
        x_train.append(data[day - timesteps:day])
        y_train.append(data[day:day+days_to_predict])

    for day in range(train_length, length-days_to_predict):
        x_test.append(data[day - timesteps:day])
        y_test.append(data[day:day+days_to_predict])

    return np.array(x_train), np.array(y_train), np.array(x_test), np.array(y_test)

# Prepare Data
timesteps = 60
days_to_predict = 3
price_scaler = MinMaxScaler(feature_range=(0, 1))
data = np.genfromtxt("Data\TSLA3.csv", delimiter=",", usecols=[4])

data = price_scaler.fit_transform(np.reshape(data, (-1, 1))).reshape(1, -1)[0]
x_train, y_train, x_test, y_test = split_train_test(data,
                                                    timesteps=timesteps,
                                                    length=len(data),
                                                    train_length=2000,
                                                    days_to_predict=days_to_predict)
x_train, x_test = np.expand_dims(x_train, 2), np.expand_dims(x_test, 2)

model = Sequential()
#model.add(LSTM(units=256, input_shape=(timesteps, 1)))
model.add(LSTM(units=256, input_shape=(timesteps, 1), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(units=256))
model.add(Dense(days_to_predict))

model.compile(optimizer='adam', loss='mse', metrics=['mse'])
history = model.fit(x_train, y_train, epochs=50)

```

```
plt.plot(history.history['mse'])
plt.show()

plt.figure(figsize=(12, 8))

predicted_price = model.predict(x_test)

plt.plot(price_scaler.inverse_transform(y_test[::days_to_predict].reshape(-1, 1)),
label="Actual")
plt.plot(price_scaler.inverse_transform(predicted_price[::days_to_predict].reshape(-1, 1)),
label="Predicted")
plt.legend()
plt.show()
```