Якщо ви у пошуках посібника по штучним нейронним мережам (ШНМ), то, можливо, ви вже маєте припущення щодо того, що це таке. Але чи знали ви, що нейронні мережі – основа нової та цікавої області, глибинного навчання? Глибинне навчання – область машинного навчання, яка у наші часи допомогла зробити великий прорив у багатьох речах, починаючи з гри в Го та Покер з живими гравцями, та закінчуючи безпілотними автомобілями. Але, насамперед, глибинне навчання потребує мати знання про роботу нейронних мереж.

У цій статті будуть представлені деякі поняття, а також трохи коду та математики, за допомогою яких ви зможете побудувати *та зрозуміти* прості нейронні мережі. Для ознайомлення з матеріалом потрібно мати базові знання про матриці та диференціали. Код буде написано мовою програмування Пайтон з використанням бібліотеки numpy. Ви побудуєте ШНМ, використовуючи Пайтон, яка з високою точністю класифікуватиме числа на картинках .

Зміст:

---

1 Що таке штучна нейронна мережа?

Штучні нейронні мережі (ШНМ) - це програмна імплементація нейронних структур нашого мозку. Ми не будемо обговорювати складну біологію нашої голови, досить знати, що мозок містить *нейрони*, які є свого роду **органічними перемикачами**. Вони можуть змінювати тип переданих сигналів в залежності від електричних або хімічних сигналів, які в них передаються. Нейронна мережа у людському мозку - величезна взаємопов'язана система нейронів, де сигнал, який передається одним нейроном, може передаватися у тисячі інших нейронів. *Навчання* відбувається через повторну активацію деяких нейронних з'єднань. Через це збільшується імовірність виведення потрібного результату при відповідній вхідній інформації (сигналах). Такий вид навчання використовує *зворотний зв'язок* - при правильному результаті нейронні зв'язки, які виводять його, стають більш щільними.

Штучні нейронні мережі імітують поведінку мозку у простішому вигляді. Вони можуть бути навчені *контрольованим* та *неконтрольованим* шляхами. У

контрольованій ШНМ, мережа навчається шляхом передавання відповідної вхідної інформації та прикладів вихідної інформації. Наприклад, спам-фільтр у електронній поштовій скриньці: вхідною інформацією може бути список слів, які зазвичай містяться у спам-повідомленнях, а вихідною інформацією - класифікація для відповідного повідомлення (спам, чи не спам). Такий вид навчання додає *ваги* зв'язкам ШНМ, але це буде обговорено пізніше.

Неконтрольоване навчання у ШНМ намагається "змусити" ШНМ "зрозуміти" структуру переданої вхідної інформації "самостійно". Ми не будемо розглядати це у даному пості.

---

2 Структура ШНМ

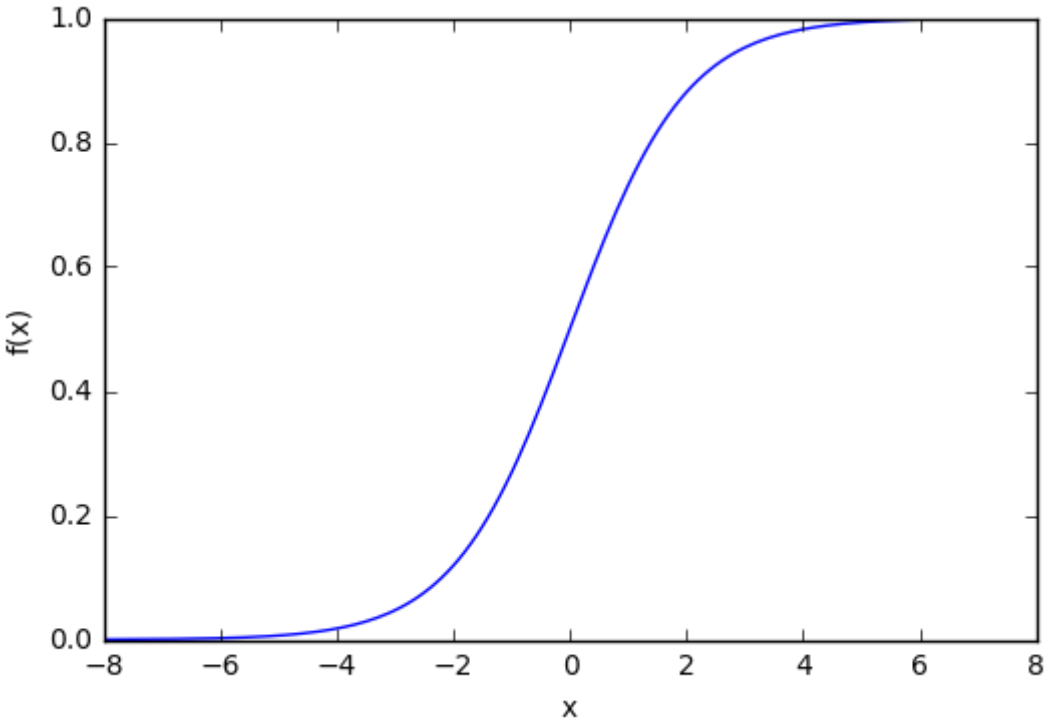
2.1 Штучний нейрон

Біологічний нейрон імітується у ШНМ через *активаційну функцію*. У задачах класифікації (наприклад визначення спам-повідомлень) активаційна функція повинна мати характеристику "вмикача". Іншими словами, якщо вхід більше, ніж деяке значення, то вихід повинен змінювати стан, наприклад з 0 на 1 або -1 на 1. Це імітує "включення" біологічного нейрону. У якості активаційної функції зазвичай використовують сигмоїдну функцію:

$$f(z) = \frac{1}{1 + exp(-z)}$$

Яка виглядає наступним чином:

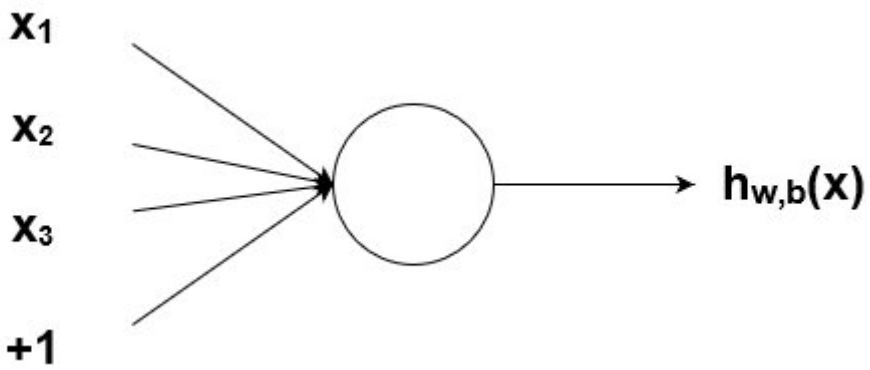
```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1 / (1 + np.exp(-x))
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```



З графіку можна побачити, що функція "активаційна" – вона росте з 0 до 1 з кожним збільшенням значення x. Сигмоїдна функція є гладкою і неперервною. Це означає, що функція має похідну, що у свою чергу є дуже важливим фактором для навчання алгоритму.

2.2 Вузли

Як було згадано раніше, біологічні нейрони ієрархічно з'єднані в мережамі, де вихід одних нейронів є входом для інших нейронів. Ми можемо представити такі мережі у вигляді з'єднаних шарів з *вузлами*. Кожен вузол приймає зважений вхід, активує *активаційну функцію* для суми входів, та генерує вихід.



Вузли зі входами

Коло на картинці зображує вузол. Вузол є "місцерозташуванням" активаційної функції, він приймає зважені входи, сумує їх, а потім вводить їх в активаційну функцію. Вивід активаційної функції представлений через h. Примітка: у деякій літературі *вузол* також називають *персептроном*.

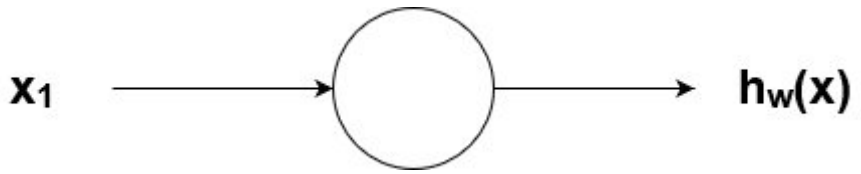
Що є "вагою"? За вагу беруться числа (не бінарні), які потім множаться на вході і сумуються у вузлі. Іншими словами, зважений вхід у вузол має вигляд:

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

Де w_i - числові значення ваги (b ми будемо обговорювати пізніше). Ваги нам потрібні, вони є значеннями, які будуть змінюватись протягом процесу навчання. b є вагою елемента зміщення на +1, включення ваги b робить вузол гнучкішим. Простіше це зрозуміти на прикладі.

2.3 Зміщення

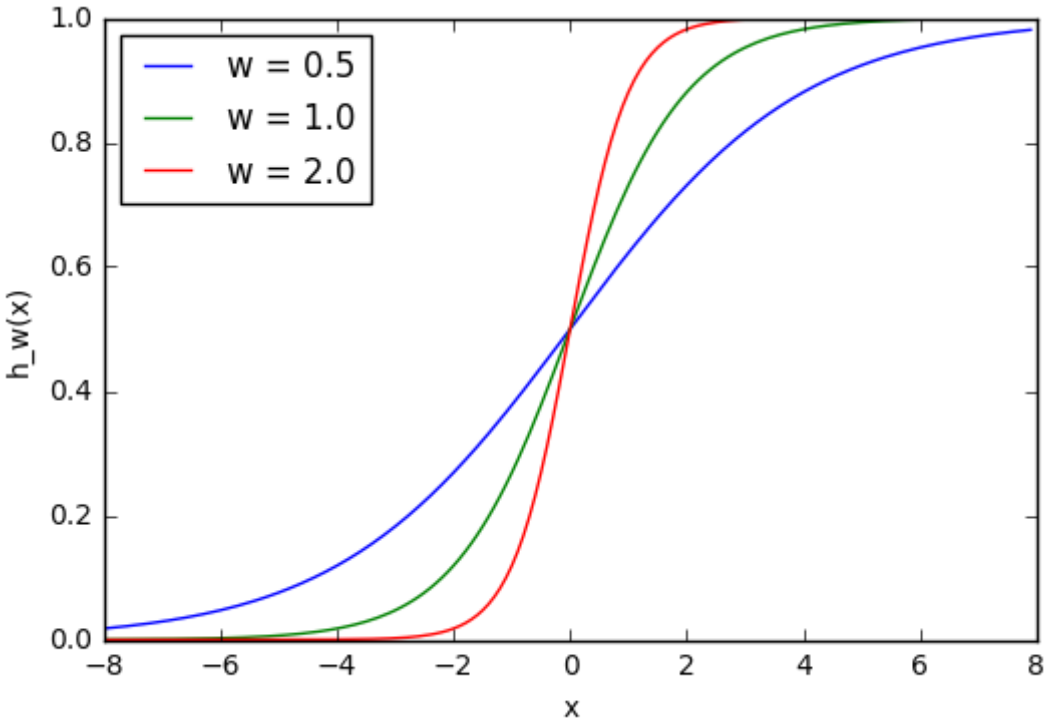
Роглянемо простий вузол, у якому є по одному входу та виходу:



Простий вузол

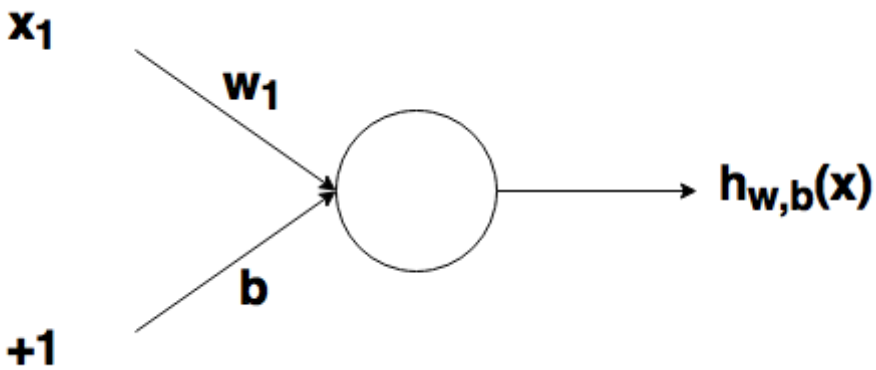
Вводом для активаційної функції у цьому вузлі є просто $x_1 w_1$. На що впливає зміна в w_1 у цій простій мережі?

```
w1 = 0.5
w2 = 1.0
w3 = 2.0
l1 = 'w = 0.5'
l2 = 'w = 1.0'
l3 = 'w = 2.0'
for w, l in [(w1, l1), (w2, l2), (w3, l3)]:
    f = 1 / (1 + np.exp(-x*w))
    plt.plot(x, f, label=l)
plt.xlabel('x')
plt.ylabel('h_w(x)')
plt.legend(loc=2)
plt.show()
```



Ефект від збільшення ваги

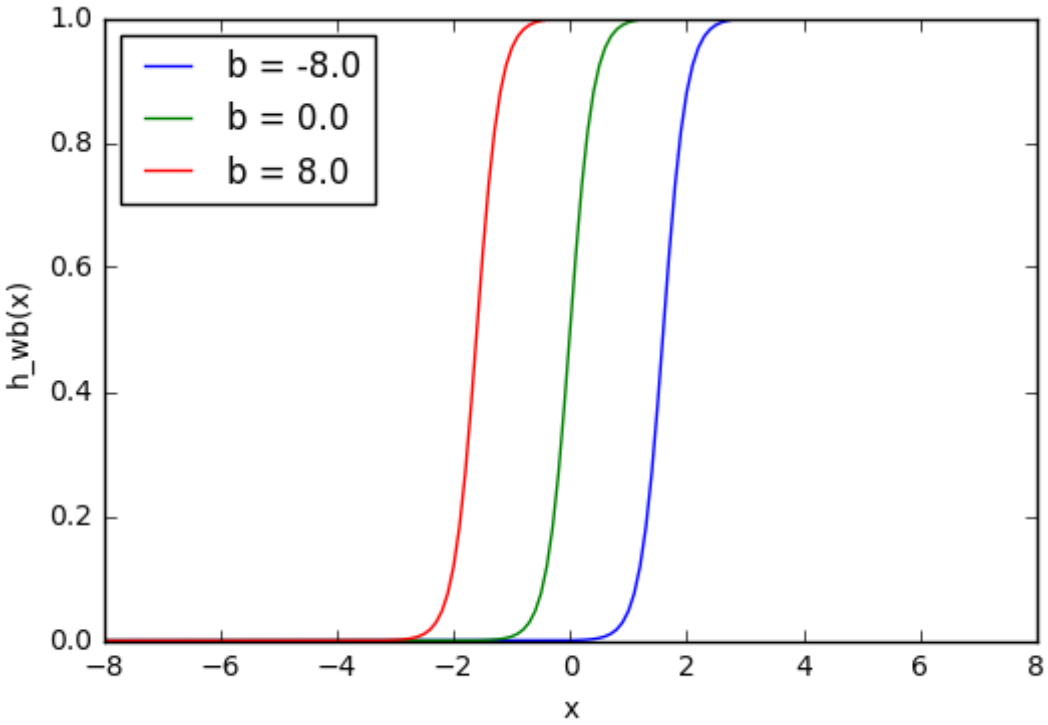
Тут ми можемо бачити, що при зміні ваги змінюється також рівень нахилу графіка активаційної функції. Це корисно, якщо ми моделюємо різні щільності взаємозв'язків між входами та виходами. Але що робити, якщо ми хочемо, щоб вихід змінювався тільки при x більше 1? Для цього нам потрібне зміщення. Розглянемо таку мережу із зміщенням на вході:



Ефект зміщення

```
w = 5.0
b1 = -8.0
b2 = 0.0
b3 = 8.0
l1 = 'b = -8.0'
l2 = 'b = 0.0'
l3 = 'b = 8.0'
for b, l in [(b1, l1), (b2, l2), (b3, l3)]:
    f = 1 / (1 + np.exp(-(x*w+b)))
    plt.plot(x, f, label=l)
plt.xlabel('x')
plt.ylabel('h_wb(x)')
```

```
plt.legend(loc=2)
plt.show()
```

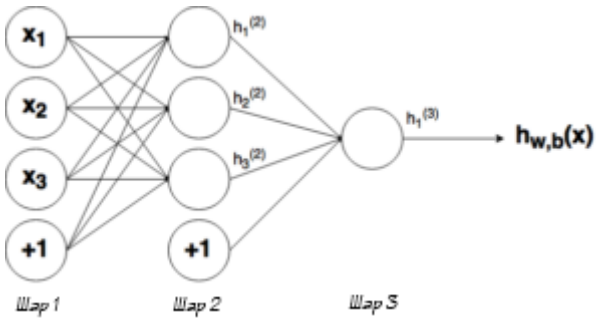


Ефект зміщення

З графіку можна побачити, що змінюючи "вагу" зміщення b, ми можемо змінювати час запуску вузла. Зміщення дуже важливе у випадках, коли потрібно імітувати умовні відносини.

2.4 Складена структура

Вище було пояснено, як працює відповідний вузол/нейрон/персептрон. Але, як ви вже знаєте, у повній нейронній мережі знаходиться багато таких взаємозв'язаних між собою вузлів. Структури таких мереж можуть приймати міріади різних форм, але найпоширеніша складається з вхідного шару, прихованого шару та вихідного шару. Приклад такої структури приведено нижче:



Три яруси нейронної мережі

Ну рисунку вище можна побачити три шари мережі - Шар 1 є **вхідним шаром**, де мережа приймає зовнішні вхідні дані. Шар 2 називають **прихованим шаром**, цей шар не є частиною ні входу, ні виходу. *Примітка:* нейронні мережі можуть мати декілька прихованих шарів, у даному прикладі було включено лише один шар для простоти. І нарешті, Шар 3 є **вихідним шаром**. Ви можете помітити, що між Шаром 1(Ш1) та Шаром 2(Ш2) існує багато зв'язків. Кожен вузол у Ш1 має зв'язок зі всіма вузлами у Ш2, при цьому від кожного вузла у Ш2 йде по одному зв'язку до єдиного вихідного вузла у Ш3. Кожен з цих зв'язків повинен мати відповідну вагу.

2.5 Позначення

Вся математика, приведена вище, потребує дуже точної нотації. Нотація, яка використовується тут, використовується і в посібнику по глибинному навчанню від Стенфордського Університету. У наступних рівняннях вага відповідного зв'язка буде позначатися як $w_{ij}^{(l)}$, де i - номер вузла у шарі $l+1$, а j - номер вузла у шарі l. Наприклад, вага зв'язка між вузлом 1 у шарі 1 та вузлом 2 у шарі 2 буде позначатися як $w_{21}{}^{(1)}$. Дивно, чому індекси 2-1 означають зв'язок 1-2? Така нотація більш зрозуміла, якщо додати зміщення.

З графу вище видно, що зміщення +1 зв'язане з усіма вузлами у сусідньому шарі. Зміщення у Ш1 має зв'язок зі всіма вузлами у Ш2. Так як зміщення не є справжнім вузлом з активаційною функцією, воно не має і входів (його вхідне значення завжди дорівнює константі). Вагу зв'язка між зміщенням і вузлом будемо позначати через $b_i^{(l)}$, де i - номер вузла у шарі $l+1$, так само, як у $w_{ij}^{(l)}$. До прикладу з $w_{21}{}^{(1)}$, вага між зміщенням у Ш1 та другим вузлом у Ш2 буде мати позначення $b_2{}^{(1)}$.

Пам'ятайте, що ці значення - $w_{ij}^{(l)}$ та $b_i^{(l)}$ - будуть змінюватись протягом процесу навчання ШНМ.

Позначення зв'язку до вихідного вузла буде виглядати наступним чином: $h_j^{(l)}$, де j - номер вузла у шарі l. Тоді у попередньому прикладі, зв'язком до вихідного вузла є $h_1{}^{(2)}$.

Тепер давайте розглянемо, як розраховувати вихід мережі, коли нам відомі вага та вхід. Процес знаходження виходу у нейронній мережі називається процесом *прямого поширення*.

---

3 Процес прямого поширення

Щоб продемонструвати, як знаходити вихід, маючи вже відомий вхід, у нейронних мережах, почнемо з попереднього прикладу з трьома шарами. Нижче така система представлена у вигляді системи рівнянь:

$$
\begin{aligned}
h_1^{(2)} &= f(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)}) \\
h_2^{(2)} &= f(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)}) \\
h_3^{(2)} &= f(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b}(x) = h_1^{(3)} &= f(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)})
\end{aligned}
$$

, де $f(\bullet)$ активаційна функція вузла, у нашому випадку сигмоїдна функція. У першому рядку $h_1{}^{(2)}$ - вихід першого вузла у другому шарі, його входами відповідно є $w_{11}^{(1)}x_1$, $w_{12}^{(1)}x_2$,$w_{13}^{(1)}x_3$ та $b_1^{(1)}$. Ці входи було просумовано, а потім передано в активаційну функцію для розрахунку виходу першого вузла. З двома наступними вузлами аналогічно.

Останній рядок розраховує вихід єдиного вузла в останньому третьому шарі, він є кінцевою вихідною точкою в нейронній мережі. У ньому замість зважених вхідних змінних (x_1, x_2, x_3) беруться зважені виходи вузлів з другого шару $(h_1^{(2)}, h_2^{(2)}, h_3^{(2)})$ та зміщення. Така система рівнянь також добре показує ієрархічну структуру нейронної мережі.

3.1 Приклад прямого поширення

Приведемо простий приклад першого виводу нейронної мережі мовою Пайтон. Зверніть увагу, ваги $w_{11}^{(1)}, w_{12}^{(1)}, \ldots$ між Ш1 та Ш2 ідеально можуть бути представлені на матриці:

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{pmatrix}$$

Представимо цю матрицю через масиви бібліотеки numpy.

```
import numpy as np
w1 = np.array([[0.2, 0.2, 0.2], [0.4, 0.4, 0.4], [0.6, 0.6, 0.6]])
```

Ми просто присвоїли деякі рандомні числові значення вазі кожного зв'язку з Ш1. Аналогічно можна зробити і з Ш2:

$$W^{(2)} = \begin{pmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} \end{pmatrix}$$

```
w2 = np.zeros((1, 3))
w2[0,:] = np.array([0.5, 0.5, 0.5])
```

Ми також можемо присвоїти деякі значення вазі зміщення у Ш1 та Ш2:

```
b1 = np.array([0.8, 0.8, 0.8])
b2 = np.array([0.2])
```

Нарешті, перед написанням основної програми для розрахунку виходу нейронної мережі, напишемо окрему функцію для активаційної функції:

```
def f(x):
    return 1 / (1 + np.exp(-x))
```

3.2 Перша спроба реалізувати процес прямого поширення

Приведемо простий спосіб розрахунку виходу нейронної мережі, використовуючи вкладені цикли у Пайтоні. Пізніше ми швидко розглянемо більш ефективні способи.

```
def simple_looped_nn_calc(n_layers, x, w, b):
    for l in range(n_layers-1):
        #Формується вхідний масив - перемноження ваг у кожному шарі
        #Якщо перший шар, то вхідний масив дорівнює вектору x
        #Якщо шар не перший, вхід для поточного шару дорівнює
        #виходу попереднього
        if l == 0:
            node_in = x
```

```
        else:
            node_in = h
        #Формує вихідний масив для вузлів у шарі l + 1
        h = np.zeros((w[l].shape[0],))
        #проходить по рядкам масиву ваг
        for i in range(w[l].shape[0]):
            #рахує суму всередині активаційної функції
            f_sum = 0
            #проходить по стовпцям масиву ваг
            for j in range(w[l].shape[1]):
                f_sum += w[l][i][j] * node_in[j]
            #додає зміщення
            f_sum += b[l][i]
            #використовує активаційну функцію для розрахунку
            #i-того виходу, у даному випадку h1, h2, h3
            h[i] = f(f_sum)
    return h
```

Дана функція приймає у якості входу номер шару у нейронній мережі, x - вхідний масив/вектор:

```
w = [w1, w2]
b = [b1, b2]
#рандомізований вхідний вектор x
x = [1.5, 2.0, 3.0]
```

Функція спочатку перевіряє, чим є вхідний масив для відповідного шару з вузлами/вагами. Якщо розглядається перший шар, то входом для другого шару є вхідний масив x, помножений на відповідні ваги. Якщо шар не перший, то входом для наступного буде вихід попереднього.

Виклик функції:

```
simple_looped_nn_calc(3, x, w, b)
```

повертає результат 0.8354. Можна перевірити правильність, вставивши ті ж значення у систему рівнянь:

$$
\begin{aligned}
h_1^{(2)} &= f(0.2 \times 1.5 + 0.2 \times 2.0 + 0.2 \times 3.0 + 0.80) = 0.8909 \\
h_2^{(2)} &= f(0.4 \times 1.5 + 0.4 \times 2.0 + 0.4 \times 3.0 + 0.80) = 0.9677 \\
h_3^{(2)} &= f(0.6 \times 1.5 + 0.6 \times 2.0 + 0.6 \times 3.0 + 0.8) = 0.9909 \\
h_{W,b}(x) = h_1^{(3)} &= f(0.5 \times 0.8909 + 0.5 \times 0.9677 + 0.5 \times 0.9909 + 0.2) = 0.8354
\end{aligned}
$$

3.3 Більш ефективна імплементація

Використання циклів – не найефективніший способом розрахунку прямого поширення мовою Пайтон, тому що цикли у цій мові програмування працюють досить повільно. Ми коротко розглянемо кращі рішення. Також можна буде порівняти роботу алгоритмів, використавши функцію у IPython:

```
%timeit simple_looped_nn_calc(3, x, w, b)
```

У даному випадку процес прямого поширення з циклами займає близько 40 мікросекунд. Це досить швидко, але не для великих нейронних мереж з >100 вузлами на кожному шарі, особливо при їх навчанні. Якщо ми запустимо цей алгоритм на нейронній мережі з чотирма шарами, то отримаємо результат 70 мікросекунд. Ця різниця є досить значною.

3.4 Векторизація у нейронних мережах

Можна більш компактно написати попередні рівняння, тим самим знайти результат ефективніше. Спочатку додамо ще одну змінну $z_i^{(l)}$, яка є сумою входу у вузол i шару l, включаючи зміщення. Тоді для першого вузла у Ш2, z буде дорівнювати:

$$z_1^{(2)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3 + b_1^{(1)} = \sum_{j=1}^{n} w_{ij}^{(1)} x_i + b_i^{(1)}$$

, де n - кількість вузлів у Ш1. Використовуючи це позначення, систему рівнянь можна скоротити:

$$z^{(2)} = W^{(1)} x + b^{(1)}$$
$$h^{(2)} = f(z^{(2)})$$
$$z^{(3)} = W^{(2)} h^{(2)} + b^{(2)}$$
$$h_{W,b}(x) = h^{(3)} = f(z^{(3)})$$

Зверніть увагу на велику W, яка означає матричну форму представлення ваг. Пам'ятайте, що тепер усі елементи у рівнянні зверху є матрицями/векторами. Але на цьому спрощення не закінчується. Дані рівняння можна звести до ще коротшого вигляду:

$$z^{(l+1)} = W^{(l)}h^{(l)} + b^{(l)}$$
$$h^{(l+1)} = f(z^{(l+1)})$$

Так виглядає загальна форма процесу прямого поширення, до вихід шару l стає входом у шар $l + 1$. Ми знаємо, що $h_{(1)}$ є вхідним шаром x, а $h^{(n_l)}$(де n_l - номер шару у мережі) є вихідним шаром. Ми також не стали використовувати індекси i та j через те, що можна просто перемножити матриці - це дає нам той самий результат. Тому даний процес і називається "векторизацією". Даний метод має ряд плюсів. По-перше, його код імплементації виглядає менш заплутаним. По-друге, використовуються властивості з лінійної алгебри замість циклів, що робить роботу програми швидшою. З numpy можна легко зробити такі підрахунки. У наступній частині швидко повторимо операції над матрицями, для тих, хто їх трохи підзабув.

3.5 Множення матриць

Розпишемо $z^{(l+1)} = W^{(l)}h^{(l)} + b^{(l)}$ на вираз із матриці та векторів з вхідного шару ($h^{(l)} = x$):

$$\begin{aligned}
z^{(2)} &= \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{pmatrix} \\
&= \begin{pmatrix} w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 \\ w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 \\ w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 \end{pmatrix} + \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{pmatrix} \\
&= \begin{pmatrix} w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)} \\ w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)} \\ w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)} \end{pmatrix}
\end{aligned}$$

Для тих, хто не знає або забув, як перемножуються матриці, швидко згадаємо це. Коли матриця ваг множиться на вектор, кожний елемент у **рядку** матриці ваг множиться на кожний елемент у **стовпці** вектора, після цього усі добутки сумуються і створюється новий вектор (3x1). Після перемноження матриці на вектор, додаються елементи з вектора зміщення та отримується кінцевий результат.

Кожний рядок отриманого вектора відповідає аргументу активаційної функції в оригінальній не матричній системі рівнянь вище. Це означає, що у Пайтоні ми можемо реалізувати все, не використовуючи повільні цикли. На щастя, бібліотека numpy дає можливість зробити це з достатньо швидкими функціями-операторами над матрицями. Розглянемо код простої та швидкої версії функції simple_looped_nn_calc:

```
def matrix_feed_forward_calc(n_layers, x, w, b):
    for l in range(n_layers-1):
        if l == 0:
            node_in = x
        else:
            node_in = h
        z = w[l].dot(node_in) + b[l]
        h = f(z)
    return h
```

Зверніть увагу на рядок 7, на якому відбувається перемноження матриці на вектор. Якщо ви використаєте замість функції перемноження *a.dot(b)* символ *, то вийде щось схоже на поелементне множення замість справжнього добутку матриць.

Якщо порівняти час роботи цієї функції з попередньою на простій мережі з чотирма шарами, то ми отримаємо результат лише на 24 мікросекунди менший. Але якщо збільшити кількість вузлів у кожному шарі до 100-100-50-10, то ми отримаємо значно більшу різницю. Функція з циклами у цьому випадку дає результат 41 мілісекунди, коли у функції з векторизацією це займає лише 84 мікросекунди. Також існують ще ефективніші імплементації операцій над матрицями, які використовують пакети глибинного навчання, такі як TensorFlow та Theano.

На цьому все про процес прямого поширення у нейронних мережах. У наступних розділах ми поговоримо про способи навчання нейронних мереж, використовуючи градієнтний спуск та зворотне поширення.

---

4 Градієнтний спуск та оптимізація

Розрахунки значень ваг, які з'єднують шари у мережі, і є тим, що ми називаємо навчанням системи. У контрольованому навчанні ідея полягає у тому, щоб зменшити похибку між входом та потрібним виходом. Якщо ми маємо нейронну мережу з одним вихідним шаром та деякий вхід x і ми хочемо, щоб на виході було число 2, але мережа видає 5, то знаходження похибки виглядає як $abs(2-5)=3$. Говорячи мовою математики, ми знайшли норму помилки L^1 (це буде обговорено пізніше).

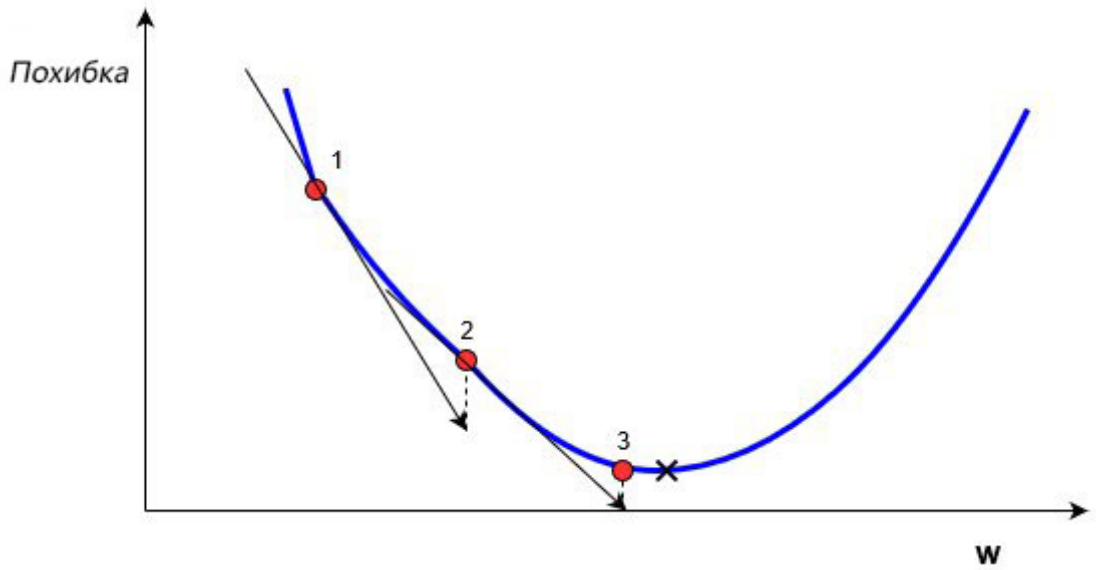
Сенс контрольованого навчання у тому, що надається багато пар вхід-вихід вже відомих даних і потрібно змінювати значення ваг, базуючись на цих прикладах, щоб значення помилки стала мінімальною. Ці пари входу-виходу позначаються як $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$, де m є кількістю екземплярів для навчання. Кожне значення входу або виходу може представляти собою вектор значень, наприклад $x^{(1)}$ не обов'язково має лише одне значення, воно може містити N-вимірний набір значень. Припустимо, що ми навчаємо нейронну мережу виявленню спам-повідомлень - у такому випадку $x^{(1)}$ може представляти собою кількість відповідних слів, які зустрічаються у повідомленні:

$$x^{(1)} = \begin{pmatrix} No.\ of\text{“}prince\text{”} \\ No.\ of\text{“}nigeria\text{”} \\ No.\ of\text{“}extension\text{”} \\ \vdots \\ No.\ of\text{“}mum\text{”} \\ No.\ of\text{“}burger\text{”} \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

$y^{(1)}$ у цьому випадку може представляти собою єдине скалярне значення, наприклад, 1 або 0, що позначає, було повідомлення спамом чи ні. У інших застосунках це також може бути вектор з K вимірами. Наприклад, ми маємо вхід x, який є вектором чорно-білих пікселів, зчитаних з фотографії. При

цьому y може бути вектором з 26 елементами із значеннями 1 або 0, що позначають, яка літера була зображена на фото, наприклад $(1, 0, \ldots, 0)$ для літери а, $(0, 1, \ldots, 0)$ для літери б і т.д.

У навчанні мережі, використовуючи (x, y), метою є покращувати її у знаходженні правильного y при відомому x. Це робиться через зміну значень ваг, щоб мінімізувати похибку. Як тоді змінювати їх значення? Для цього нам і знадобиться **градієнтний спуск**. Розглянемо наступний графік:



Одновимірний градієнтний спуск

На цьому графіку зображено похибку, залежну від скалярного значення ваги, w. Мінімально можлива похибка позначена чорним хрестом, але ми не знаємо яке саме значення w дає нам це мінімальне значення. Підрахунок починається з рандомного значення змінної w, яка дає похибку, позначену червоною крапкою під номером "1" на кривій. Нам потрібно змінити w таким чином, щоб досягнути мінімальної похибки, чорного хреста. Одним з найпоширеніших способів є градієнтний спуск.

Спочатку знаходиться градієнт похибки на "1" по відношенню до w. *Градієнт* є рівнем нахилу кривої у відповідній точці. Він зображений на графіку у вигляді чорних стрілок. Градієнт також дає деяку інформацію про напрямок - якщо він позитивний при збільшенні w, то у цьому *напрямку* похибка буде збільшуватись, якщо негативний - зменшуватись(див. графік). Як ви вже зрозуміли, ми намагаємося зробити, щоб похибка з кожним кроком

зменшувалась. Величина *градієнта* означає те, як швидко крива похибки або функція змінюється у відповідній точці. Чим більше значення, тим швидше змінюється похибка у відповідній точці у залежності від w.

Метод градієнтного спуску використовує градієнт, щоб приймати рішення щодо наступної зміни у w для того, щоб досягнути мінімального значення кривої. Він є ітеративним методом, кожен раз оновлюється значення w через:

$$w_{\text{н}} = w_{\text{ст}} - \alpha * \nabla error$$

, де $w_{\text{н}}$ означає нове значення w, $w_{\text{ст}}$ - поточне або "старе" значення w, $\nabla error$ є градієнтом похибки на $w_{\text{ст}}$ та α є кроком. Крок α також буде означати, як швидко відповідь наближається до мінімальної похибки. При кожній ітерації у такому алгоритмі градієнт повинен зменшуватись. З графіку вище ви можете помітити, що з кожним кроком градієнт "стихає". Як тільки відповідь досягне мінімального значення, ми виходимо із ітеративного процесу. Вихід можна реалізувати способом умови "якщо похибка менше деякого числа". Це число називають *точністю*.

4.1 Простий приклад на коді

Розглянемо приклад простої імплементації градієнтного спуску для знаходження мінімуму функції $f(x) = x^4 - 3x^3 + 2$ мовою Пайтон. Градієнт цієї функції можна знайти аналітично через похідну $f'(x) = 4x^3 - 9x^2$. Це означає, що для будь-якого x ми можемо знайти градієнт за цією простою формулою. Ми також можемо знайти мінімум через похідну - $x = 2.25$.

```
x_old = 0 # Немає різниці, яке значення, головне abs(x_new - x_old) > точність
x_new = 6 # Алгоритм починається з x=6
gamma = 0.01 # Розмір кроку
precision = 0.00001 # Точність

def df(x):
    y = 4 * x**3 - 9 * x**2
    return y

while abs(x_new - x_old) > precision:
    x_old = x_new
    x_new += -gamma * df(x_old)
```

```
print("Локальний мінімум знаходиться на %f" % x_new)
```

Ця функція виводить "Локальний мінімум знаходиться на 2.249965", що задовольняє правильній відповіді з деякою точністю. Цей код імплементує алгоритм зміни ваги, про яку розповідалося вище, і може знаходити мінімум функції з відповідною точністю. Це був дуже простий приклад градієнтного спуску, знаходження градієнту при навчанні нейронної мережі виглядає трохи інакше, хоч і головна ідея залишається та ж сама - ми знаходимо градієнт нейронної мережі і змінюємо ваги на кожному кроці, щоб наблизитись до мінімальної похибки, яку ми намагаємось знайти. Але у випадку ШНМ нам потрібно буде реалізувати градієнтний спуск з багатовимірним вектором ваг.

Ми будемо знаходити градієнт нейронної мережі, використовуючи досить популярний метод зворотного поширення похибки, про який буде написано пізніше. Але спочатку нам потрібно розглянути функцію похибки більш детально.

4.2 Функція оцінки

Існує більш загальний спосіб зобразити вирази, які дають нам можливість зменшити похибку. Таке загальне представлення має назву **функція оцінки**. Наприклад, функція оцінки для пари вхід-вихід (x^z, y^z) у нейронній мережі буде виглядати наступним чином:

$$\begin{aligned} J(w,b,x,y) &= \frac{1}{2} \| y^z - h^{(n_l)}(x^z) \|^2 \\ &= \frac{1}{2} \| y^z - y_{\text{п}}(x^z) \|^2 \end{aligned}$$

Вираз є функцією оцінки навчального екземпляра z_{th}, де $h^{(n_l)}$ є виходом останнього шару, тобто вихід нейронної мережі. $h^{(n_l)}$ також можна представити як $y_{\text{п}}$, що означає отриманий результат, коли нам відомий вхід x^z. Дві вертикальні лінії означають норму L^2 похибки або суму квадратів похибок. Сума квадратів похибок є досить поширеним способом представлення похибок у системі машинного навчання. Замість того, щоб

брати абсолютну похибку $abs(y_{pred}(x^z) - y^z)$, ми беремо квадрат похибки. Ми не будемо обговорювати причину цього у даній статті. $\frac{1}{2}$ у початку є просто константою, яка нормалізує відповідь після того, як ми продиференціюємо функцію оцінки під час зворотного поширення.

Зверніть увагу, що приведена раніше функція оцінки працює тільки з однією парою (x, y). Ми хочемо мінімізувати функцію оцінки з усіма m парами вхід-вихід:

$$J(w,b) = \frac{1}{m}\sum_{z=0}^{m}\frac{1}{2} \parallel y^z - h^{(n_l)}(x^z)\parallel^2$$
$$= \frac{1}{m}\sum_{z=0}^{m} J(W,b,x^{(z)},y^{(z)})$$

Тоді як же ми будемо використовувати функцію J для навчання наших мереж? Звичайно, використовуючи градієнтний спуск та зворотне поширення похибок. Спочатку розглянемо градієнтний спуск у нейронних мережах більш детально.

4.3 Градієнтний спуск у нейронних мережах

Градієнтний спуск для кожної ваги $w_{(ij)}^{(l)}$ та зміщення $b_i^{(l)}$ у нейронній мережі виглядає наступним чином:

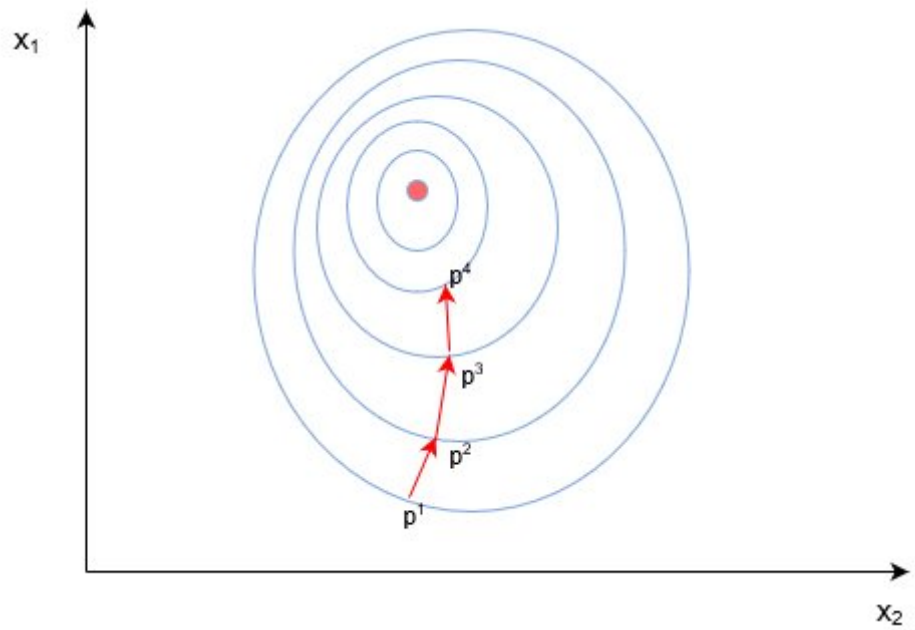
$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha\frac{\partial}{\partial w_{ij}^{(l)}}J(w,b)$$
$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}J(w,b)$$

Вираз вище фактично є аналогічним представленню градієнтного спуску: $w_{new} = w_{old} - \alpha * \nabla error$. Немає лише деяких позначень, але достатньо розуміти, що ліворуч розташовані *нові* значення, а праворуч - *старі*. Знову ж таки задіяно ітераційний процес для розрахунку ваг на кожній ітерації, але цього разу базуючись на функції оцінки $J(w, b)$.

Значення $\frac{\partial}{\partial w_{ij}^{(l)}}$ та $\frac{\partial}{\partial b_i^{(l)}}$ є *частковими похідними* функції оцінки, базуючись на значеннях ваги. Що це означає? Згадайте простий приклад градієнтного спуску раніше, кожний крок залежить від *нахилу* похибки/оцінки по відношенню до ваги. *Похідна* також має значення нахилу/градієнту. Звичайно, похідна позначається як $\frac{d}{dx}$. x у нашому випадку є вектором, а це значить, що наша похідна теж буде вектором, який є градієнт кожного виміру x.

4.4 Приклад двовимірного градієнтного спуску

Розглянемо приклад стандартного двовимірного градієнтного спуску. Нижче представлено діаграму роботи двох ітеративних двовимірних градієнтних спусків:



Двовимірний градієнтний спуск

Синім позначені контури функції оцінки, вони позначають області, у яких значення похибки приблизно однакові. Кожен крок ($p_1 \rightarrow p_2 \rightarrow p_3$) у градієнтному спуску використовує градієнт або похідну, що позначається стрілкою/вектором. Цей вектор проходить через два простори $[x_1, x_2]$ і показує напрямок, у якому знаходиться мінімум. Наприклад, похідна, обчислена у p_1 може бути $\frac{d}{dx} = [2.1, 0.7]$, де похідна є вектором з двома значеннями. Часткова похідна $\frac{\partial}{\partial x_1}$ у цьому випадку дорівнює скаляру $\rightarrow [2.1]$ -

іншими словами, це є значення градієнта лише у одному вимірі пошукового простору (x_1).

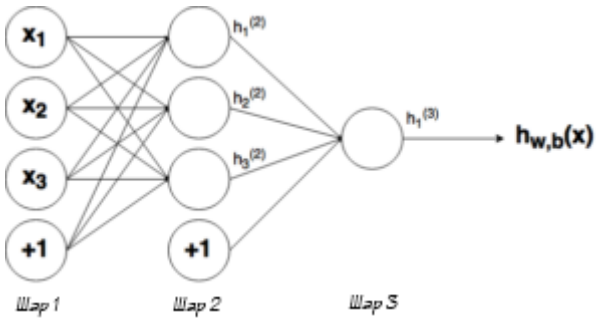
У нейронних мережах не існує простої повної функції оцінки, з якої можна легко порахувати градієнт, схожої на функцію, яку ми раніше розглядали ($f(x) = x^4 - 3x^3 + 2$). Ми можемо порівняти вихід нейронної мережі з нашим очікуваним значенням $y^{(z)}$, після чого функція оцінки буде змінюватись через зміну у значеннях ваги, але як ми це зробимо з усіма *прихованими* шарами у мережі?

Тому нам потрібен метод зворотного поширення. Цей метод дає нам можливість "ділити" функцію оцінки або похибку з усіма вагами у мережі. Іншими словами, ми можемо з'ясувати, як впливає кожна вага на похибку.

4.5 Зворотне поширення вглиб

Якщо математика вам не дуже добре дається, то ви можете пропустити цей розділ. У наступному розділі ви дізнаєтесь, як реалізувати зворотне поширення мовою програмування. Але якщо ви не проти трохи більше поговорити про математику, то продовжуйте читати, ви отримаєте більш глибокі знання щодо навчання нейронних мереж.

Спочатку, давайте згадаємо базові рівняння для нейронної мережі з трьома шарами з попередніх розділів:



Вихід цієї нейронної мережі знаходиться за формулою:

$$h_{W,b}(x) = h_1^{(3)} = f(w_{11}^{(2)} h_1^{(2)} + w_{12}^{(2)} h_2^{(2)} + w_{13}^{(2)} h_3^{(2)} + b_1^{(2)})$$

Ми також можемо спростити це рівняння до $h_1^{(3)} = f(z_1^{(2)})$, додавши нове значення $z_1^{(2)}$, яке означає:

$$z_1^{(2)} = w_{11}^{(2)} h_1^{(2)} + w_{12}^{(2)} h_2^{(2)} + w_{13}^{(2)} h_3^{(2)} + b_1^{(2)}$$

Припустимо, що ми хочемо дізнатись, як впливає зміна у вазі $w_{12}^{(2)}$ на функцію оцінки. Це означає, що нам потрібно обчислити $\frac{\partial J}{\partial w_{12}^{(2)}}$. Щоб зробити це, потрібно використати правило диференціювання складної функції:

$$\frac{\partial J}{\partial w_{12}^{(2)}} = \frac{\partial J}{\partial h_1^{(3)}} \frac{\partial h_1^{(3)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}}$$

Якщо придивитися, то права частина повністю скорочується (по принципу $\frac{2}{5}\frac{5}{2} = \frac{2}{2} = 1$). $\frac{\partial J}{\partial w_{12}^{(2)}}$ було розбито на три множники, два з яких можна чудово замінити. Почнемо з $\frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}}$:

$$\begin{aligned} \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}} &= \frac{\partial}{\partial w_{12}^{(2)}} (w_{11}^{(1)} h_1^{(2)} + w_{12}^{(1)} h_2^{(2)} + w_{13}^{(1)} h_3^{(2)} + b_1^{(1)}) \\ &= \frac{\partial}{\partial w_{12}^{(2)}} (w_{12}^{(1)} h_2^{(2)}) = h_2^{(2)} \end{aligned}$$

Часткова похідна $z_1^{(2)}$ по $w_{12}^{(2)}$ залежить тільки від одного добутку у дужках, $w_{12}^{(1)} h_2^{(2)}$, так як усі елементи у дужках, крім $w_{12}^{(2)}$, не змінюються. Похідна від константи завжди дорівнює 1, а $\frac{\partial}{\partial w_{12}^{(2)}} (w_{12}^{(1)} h_2^{(2)})$ скорочується до просто $h_2^{(2)}$, що є звичайних виходом другого вузла з шару 2.

Наступна часткова похідна складної функції $\frac{\partial h_1^{(3)}}{\partial z_1^{(2)}}$ є частковою похідною активаційної функції вихідного вузла $h_1^{(3)}$. Через те, що нам потрібно брати похідні активаційної функції, випливає основна умова її включення у нейронні мережі - функція повинна бути *диференційованою*. Для сигмоїдної активаційної функції похідна буде виглядати так:

$$\frac{\partial h}{\partial z} = f'(z) = f(z)(1 - f(z))$$

, де $f(z)$ є самою активаційною функцією. Тепер нам потрібно розібратись, що робити з $\frac{\partial J}{\partial h_1^{(3)}}$. Згадайте, що $J(w, b, x, y)$ є функція квадрату похибки, яка виглядає так:

$$J(w, b, x, y) = \frac{1}{2} \parallel y_1 - h_1^{(3)}(z_1^{(2)}) \parallel^2$$

Тут y_1 є очікуваним виходом для вихідного вузла. Знову використаємо правило диференціювання складної функції:

$$\begin{aligned}
&Let\ u = \parallel y_1 - h_1^{(3)}(z_1^{(2)}) \parallel \text{ та } J = \frac{1}{2}u^2 \\
&\text{Використовуючи } \frac{\partial J}{\partial h} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial h} : \\
&\frac{\partial J}{\partial h} = -(y_1 - h_1^{(3)})
\end{aligned}$$

Ми з'ясували, як знаходити $\frac{\partial J}{\partial w_{12}^{(2)}}$ принаймні для ваг зв'язків з вихідним шаром. Перед тим, як перейти до одного з прихованих шарів, введемо деякі нові значення δ, щоб трохи скоротити наші вирази:

$$\delta_i^{(n_l)} = -(y_i - h_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

, де i є номером вузла у вихідному шарі. У нашому прикладі є лише один вузол, тому $i = 1$. Напишемо повний вигляд похідної функції оцінки:

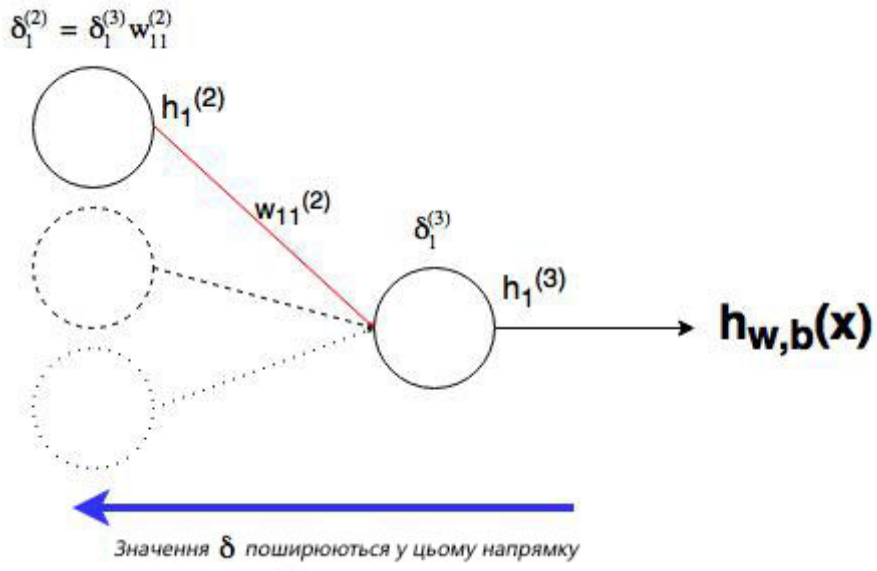
$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b, x, y) = h_j^{(l)} \delta_i^{(l+1)}$$

, де вихідний шар, у нашому випадку, $l = 2$, а i відповідає номеру вузла.

4.6 Поширення у приховані шари

Що робити з вагами у прихованих шарах(у нашому випадку у шарі 2)? Для ваг, які з'єднані з вихідним шаром, похідна $\frac{\partial J}{\partial h} = -(y_i - h_i^{(n_l)})$ мала зміст, т.я. функція оцінки може бути зразу знайдена через порівняння вихідного шару з існуючими даними. Але виходи прихованих вузлів не мають подібних вже існуючих даних для перевірки, вони зв'язані з функцією оцінки лише через інші шари вузлів. Як ми можемо знайти зміни у функції оцінки через зміни ваг, які знаходяться глибоко у нейронній мережі? Як було вже сказано, ми використовуємо метод *зворотного поширення*.

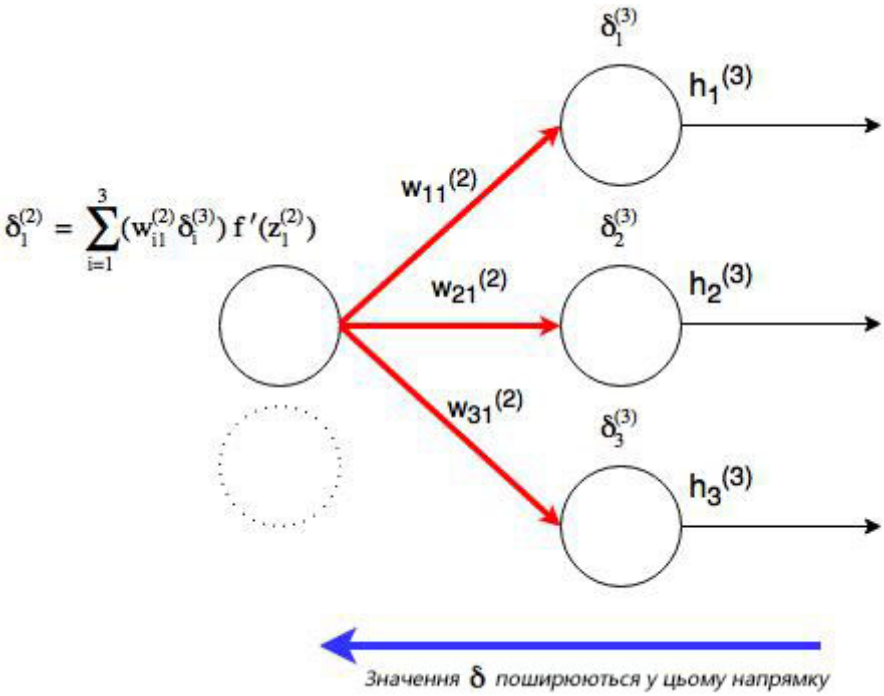
Ми вже зробили найважчу роботу з правилом диференціювання складних функцій, тепер розглянемо усе більш графічно. Значення, яке буде зворотньо поширюватись, - $\delta_i^{(n_l)}$, т.я. це є найближчим зв'язком із функцією оцінки. А що з вузлом j у другому шарі (прихованому шарі)? Як він впливає на $\delta_i^{(n_l)}$ у нашій мережі? Він змінює інші значення через вагу $w_{ij}^{(2)}$ (див. діаграму нижче, де $j = 1$ $i = 1$).



Як можна бачити з малюнка, вихідний шар *з'єднується* з прихованим вузлом через вагу. У випадку, коли у вихідному шарі є тільки один вузол, загальний вираз прихованого шару буде виглядати так:

$$\delta_j^{(l)} = \delta_1^{(l+1)} w_{1j}^{(l)} f'(z_j)^{(l)}$$

, де j номер вузла у шарі l. Але що буде, якщо у вихідному шарі знаходиться багато вихідних вузлів? У цьому випадку $\delta_j^{(l)}$ знаходиться через зважену суму всіх зв'язаних між собою похибок, як показано на діаграмі нижче:



На рисунку показано, що кожне значення δ з вихідного шару сумується для знаходження $\delta_1^{(2)}$, але кожний вихід δ повинен бути зваженим відповідними значенням $w_{i1}^{(2)}$. Іншими словами, вузол 1 у шарі 2 сприяє змінам похибок у трьох вихідних вузлах, при цьому отримана похибка (або значення функції оцінки) у кожному з цих вузлів має бути "передана назад" значенню δ цього вузла. Сформуємо загальний вираз значення δ для вузлів у прихованому шарі:

$$\delta_j^{(l)} = (\sum_{i=1}^{s_{(l+1)}} w_{ij}^{(l)} \delta_i^{(l+1)})\, f'(z_j^{(l)})$$

, де j є номером вузла у шарі l, i - номер вузла у шарі $l+1$ (що є аналогічним позначенням, яке ми використовували раніше). $s_{(l+1)}$ - це кількість вузлів у шарі $l+1$.

Тепер ми знаємо, як знаходити:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b, x, y) = h_j^{(l)} \delta_i^{(l+1)}$$

Але що робити із вагами зміщення? Принцип роботи з ними аналогічний звичайним вагам, використовуючи правила диференціювання складних функцій:

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b, x, y) = \delta_i^{(l+1)}$$

Чудово, тепер ми знаємо, як реалізувати градієнтний спуск у нейронних мережах:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial}{\partial w_{ij}^{(l)}} J(w, b)$$
$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(w, b)$$

Проте, для такої реалізації, нам потрібно буде знову застосувати цикли. Як ми вже знаємо з попередніх розділів, цикли у мові програмування Пайтон працюють досить повільно. Нам потрібно буде зрозуміти, як можна векторизувати такі підрахунки.

4.7 Векторизація зворотного поширення

Для того, щоб зрозуміти, як векторизувати процес градієнтного спуску у нейронних мережах, розглянемо спочатку спрощену векторизовану версію градієнту функції оцінки (**увага**: це поки що не є правильного версією!):

$$\frac{\partial J}{\partial W^{(l)}} = h^{(l)} \delta^{(l+1)}$$
$$\frac{\partial J}{\partial b^{(l)}} = \delta^{(l+1)}$$

Що представляє собою $h^{(l)}$? Все просто, вектор $(s_l \times 1)$, де s_l є кількістю вузлів у шарі l. Як тоді виглядає добуток $h^{(l)} \delta^{(l+1)}$? Ми знаємо, що $\alpha \times \frac{\partial J}{\partial W^{(l)}}$ має бути того самого розміру, що і матриця ваг $W^{(l)}$, ми також знаємо, що

результат $h^{(l)}\delta^{(l+1)}$ має бути того самого розміру, що і матриця ваг для шару l. Іншими словами, добуток повинен бути розміру $(s_{l+1} \times s_l)$.

Ми знаємо, що $\delta^{(l+1)}$ має розмір $(s_{l+1} \times 1)$, а $h^{(l)}$ - розмір $(s_l \times 1)$. За правилом множення матриць, якщо матрицю $(\mathbf{n} \times m)$ помножити на матрицю $(o \times \mathbf{p})$, то ми отримаємо матрицю розміру $(\mathbf{n} \times \mathbf{p})$. Якщо ми просто перемножимо $h^{(l)}$ на $\delta^{(l+1)}$, то кількість стовпців у першому векторі (один стовпець) не буде дорівнювати кількості рядків у другому веторі (3 рядки). Тому, для того, щоб можна було перемножити ці матриці та отримати результат розміру $(s_{l+1} \times s_l)$, потрібно зробити транспонування. Транспонування змінює у матриці стовпці на рядки і навпаки (наприклад матрицю виду $(s_l \times 1)$ на $(1 \times s_l)$). Воно позначається як літера T над матрицею. Ми можемо зробити наступне:

$$\delta^{(l+1)}(h^{(l)})^T = (s_{l+1} \times 1) \times (1 \times s_l) = (s_{l+1} \times s_l)$$

Використовуючи операцію транспонування, ми можемо досягти реультату, який нам потрібен.

Ще одне транспонування потрібно зробити з сумою похибок у зворотному поширенні:

$$\delta_j^{(l)} = (\sum_{i=1}^{s(l+1)} w_{ij}^{(l)} \delta_i^{(l+1)}) f'(z_j^{(l)}) = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

Символ • у попередньому виразі означає поелементне множення (добуток Адамара), що не є множенням матриць. Зверніть увагу, що добуток матриць $\left((W^{(l)})^T \delta^{(l+1)}\right)$ потребує ще зайвого підсумовування ваг та значень δ.

4.8 Імплементація етапу градієнтного спуску

Як тоді інтегрувати векторизацію у етапи градієнтного спуску нашого алгоритму? По-перше, згадаємо повний вигляд нашої функції оцінки, яку нам потрібно мінімізувати:

$$J(w,b) = \frac{1}{m}\sum_{z=0}^{m} J(W,b,x^{(z)},y^{(z)})$$

З формули видно, що повна функція оцінки складається з суми поетапних розрахунків функції оцінки. Також слід згадати, як знаходиться градієнтний спуск (поелементна та векторизована версії):

$$\begin{aligned} w_{ij}^{(l)} &= w_{ij}^{(l)} - \alpha \frac{\partial}{\partial w_{ij}^{(l)}} J(w,b) \\ W^{(l)} &= W^{(l)} - \alpha \frac{\partial}{\partial W^{(l)}} J(w,b) \\ &= W^{(l)} - \alpha \left[\frac{1}{m} \sum_{z=1}^{m} \frac{\partial}{\partial W^{(l)}} J(w,b,x^{(z)},y^{(z)}) \right] \end{aligned}$$

Це означає, що по проходженню через екземпляри навчання нам потрібно мати окрему змінну, яка дорівнюватиме сумі часткових похідних функції оцінки кожного екземпляра. Така змінна збере у собі всі значення для "глобального" підрахунку. Назвемо таку "сумуючу" змінну $\Delta W^{(l)}$. Відповідна змінна для зміщення буде позначатися як $\Delta b^{(l)}$. Отже, при кожній ітерації у процесі навчання мережі нам потрібно буде зробити наступні кроки:

$$\begin{aligned} \Delta W^{(l)} &= \Delta W^{(l)} + \frac{\partial}{\partial W^{(l)}} J(w,b,x^{(z)},y^{(z)}) \\ &= \Delta W^{(l)} + \delta^{(l+1)} (h^{(l)})^T \\ \Delta b^{(l)} &= \Delta b^{(1)} + \delta^{(l+1)} \end{aligned}$$

Виконуючи ці операції на кожній ітерації, ми підраховуємо згадану раніше суму $\sum_{z=1}^{m} \frac{\partial}{\partial W^{(l)}} J(w,b,x^{(z)},y^{(z)})$(і аналогічна формула для b). Після того, як будуть проітеровані усі екземпляри та отримані усі значення δ, ми оновлюємо значення параметрів ваги:

$$\begin{aligned} W^{(l)} &= W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)} \right] \\ b^{(l)} &= b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right] \end{aligned}$$

4.9 Кінцевий алгоритм градієнтного спуску

І, нарешті, ми дійшли до означення методу зворотного поширення через градієнтний спуск для навчання наших нейронних мереж. Фінальний алгоритм зворотношо поширення виглядає наступним чином:

Рандомно ініціалізуйте ваги для кожного шару $W^{(l)}$ Коли ітерація < границі ітерації:

1. Дайте ΔW та Δb початкове значення нуль
2. Для екземплярів від 1 до m: а. Запустіть процес прямого поширення через усі n_l шарів. Зберігайте вивід активаційної функції у $h^{(l)}$ б. Знайдіть значення $\delta^{(n_l)}$ вихідного шару в. Оновіть $\Delta W^{(l)}$ та $\Delta b^{(l)}$ для кожного шару
3. Запустіть процес градієнтного спуску, використовуючи:

$$W^{(l)} = W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

З цього алгоритму випливає, що ми будемо повторювати градієнтний спуск, поки функція оцінки не досягне мінімуму. На цьому етапі нейронна мережа вважається навченою та готовою до використання.

Далі ми спробуємо реалізувати цей алгоритм мовою програмування для навчання нейронної мережі розпізнавати числа, написані від руки.

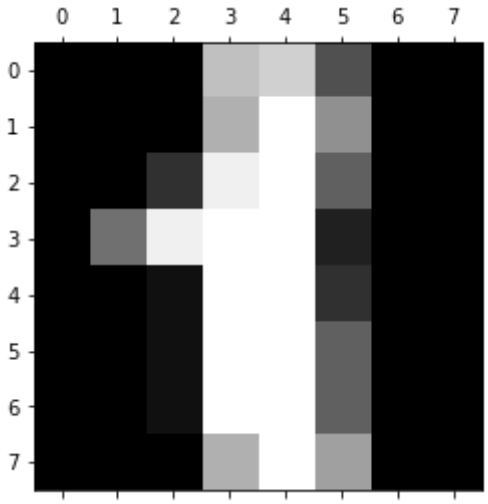
---

5 Імплементація нейронної мережі мовою Пайтон

У попередньому розділі ми розглянули теорію по навчанню нейронної мережі через градієнтний спуск та метод зворотного поширення. У цьому розділі ми використаємо набуті знання на практиці - напишемо код, який прогнозує, базуючись на даних MNIST. База даних MNIST - це збір прикладів у нейронних мережах та глибинному навчанні. Вона містить у собі зображення цифр, написаних від руки, з відповідними ярликами, які пояснюють, що це за число. Кожне зображення розміром 8x8 пікселів. У цьому прикладі ми використаємо сети даних MNIST для бібліотеки машинного навчання scikit

learn у мові програмування Пайтон. Приклад такого зображення можна побачити під кодом:

```
from sklearn.datasets import load_digits
digits = load_digits()
print(digits.data.shape)
import matplotlib.pyplot as plt
plt.gray()
plt.matshow(digits.images[1])
plt.show()
```



Код, який ми збираємось написати у нашій нейронній мережі, буде аналізувати цифри, які зображають пікселі на зображенні. Для початку, нам потрібно просортувати вхідні дані. Для цього ми зробимо дві наступні речі:

1. Промасштабувати дані
2. Розділити дані на тести та навчальні тести

5.1 Масштабування даних

Чому нам потрібно масштабувати дані? По-перше, розглянемо представлення пікселів одного із сетів даних:

```
digits.data[0,:]
Out[2]:
array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
        15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
         8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
         5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
         1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
         0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

Чи помітили ви, що вхідні дані змінюються в інтервалі від 0 до 15? Досить поширеною практикою є масштабування вхідних даних так, щоб вони були лише в інтервалі від [0, 1], або [-1, 1]. Це робиться для легшого порівняння різних типів даних у нейронній мережі. Масштабування даних можна легко зробити через бібліотеку машинного навчання scikit learn:

```
from sklearn.preprocessing import StandardScaler
X_scale = StandardScaler()
X = X_scale.fit_transform(digits.data)
X[0,:]
Out[3]:
array([ 0.        , -0.33501649, -0.04308102,  0.27407152, -0.66447751,
       -0.84412939, -0.40972392, -0.12502292, -0.05907756, -0.62400926,
        0.4829745 ,  0.75962245, -0.05842586,  1.12772113,  0.87958306,
       -0.13043338, -0.04462507,  0.11144272,  0.89588044, -0.86066632,
       -1.14964846,  0.51547187,  1.90596347, -0.11422184, -0.03337973,
        0.48648928,  0.46988512, -1.49990136, -1.61406277,  0.07639777,
        1.54181413, -0.04723238,  0.        ,  0.76465553,  0.05263019,
       -1.44763006, -1.73666443,  0.04361588,  1.43955804,  0.        ,
       -0.06134367,  0.8105536 ,  0.63011714, -1.12245711, -1.06623158,
        0.66096475,  0.81845076, -0.08874162, -0.03543326,  0.74211893,
        1.15065212, -0.86867056,  0.11012973,  0.53761116, -0.75743581,
       -0.20978513, -0.02359646, -0.29908135,  0.08671869,  0.20829258,
       -0.36677122, -1.14664746, -0.5056698 , -0.19600752])
```

Стандартний інструмент масштабування у scikit learn нормалізує дані через віднімання та ділення. Ви можете бачити, що тепер усі дані знаходяться в інтервалі від -2 до 2. Щодо вихідних даних y, то зазвичай немає необхідності їх масштабувати.

5.2 Створення тестів та навчальних наборів даних

У машинному навчанні є такая феномена, яка називається "перенавчанням". Це відбувається, коли моделі, під час навчання, стають занадто заплутаними - з ними досить добре навчені, але коли їм передаються нові дані, які вони ще ніколи на "бачили", то результат, який вони видають, виходить поганим. Іншими словами, моделі генеруються не дуже добре. Щоб впевнитись, що ми не створюємо занадто складних моделей, зазвичай розбивають набір даних у *навчальні* набори та *тестові* набори. Навчальним набором є дані, на яких модель буде навчатись, а тестовий набір - це дані, на яких модель буде тестуватись після завершення навчання. Кількість навчальних даних повинна бути завжди більше тестових даних. Зазвичай вони займають 60-80% від набору даних.

Знову ж таки, scikit learn легко розбиває дані на навчальні та тестові набори:

```
from sklearn.model_selection import train_test_split
y = digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
```

У цьому випадку ми виділили 40% даних на тестові набори, і 60% відповідно на навчання. Функція train_test_split у scikit learn закидує дані рандомно у різні бази даних - іншими словами, функція не бере перші 60% рядків у якості навчального набору, а те, що залишилося, як тестовий.

5.3 Налаштування вихідного шару

Для того, щоб отримувати результат - число від 0 до 9, нам потрібен вихідний шар. Більш-менш точна нейронна мережа, зазвичай, має вихідний шар з 10 вузлами, кожний з яких видає число від 0 до 9. Ми хочемо навчити мережу так, щоб, наприклад, при цифрі 5 на зображенні, вузол з цифрою 5 у вихідному шарі мав найбільше значення. В ідеалі, ми б хотіли мати наступний вивід: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]. Але, насправді, ми можемо отримати щось схоже на це: [0.01, 0.1, 0.2, 0.05, 0.3, 0.8, 0.4, 0.03, 0.25, 0.02]. У такому випадку ми можемо взяти найбільших індекс у вихідному масиві та вважати це нашим отриманим числом.

У даних MNIST потрібні результати від зображень записані як окреме число. Нам потрібно конвертувати це єдине число у вектор, щоб його можна було порівнювати з вихідним шаром з 10 вузлами. Іншими словами, якщо результат у MNIST позначається як "1", то нам потрібно його конвертувати у вектор: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]. Такую конвертацію імплементує наступний код:

```
import numpy as np
def convert_y_to_vect(y):
    y_vect = np.zeros((len(y), 10))
    for i in range(len(y)):
        y_vect[i, y[i]] = 1
    return y_vect


y_v_train = convert_y_to_vect(y_train)
y_v_test = convert_y_to_vect(y_test)
y_train[0], y_v_train[0]
```

```
Out[8]:
(1, array([ 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]))
```

Цей код конвертує "1" у вектор [0, 1, 0, 0, 0, 0, 0, 0, 0, 0].

5.4 Створення нейронної мережі

Наступним кроком є створення структури нейронної мережі. Для вхідного шару, ми знаємо, що нам потрібно 64 вузла, щоб покрити 64 пікселів зображення. Як було сказано раніше, нам потрібен вихідний шар з 10 вузлами. Нам також буде потрібен прихований шар у нашій мережі. Зазвичай, кількість вузлів у прихованих шарах не менше і не біьше кількості вузлів у вхідному та вихідному шарах. Оголосимо простий список мовою Пайтон, який визначає структуру нашої мережі:

```
nn_structure = [64, 30, 10]
```

Ми знову використаємо сигмоїдну активаційну функцію, так що спочатку потрібно оголосити цю функцію та її похідну:

```
def f(x):
    return 1 / (1 + np.exp(-x))

def f_deriv(x):
    return f(x) * (1 - f(x))
```

Зараз ми не маємо ніякого уявлення, як виглядає наша нейронна мережа. Як ми будемо її навчати? Згадаємо наш алгоритм з попередніх розділів:

Рандомно ініціалізуйте ваги для кожного шару $W^{(l)}$ Коли ітерація < границі ітерації:

1. Дайте ΔW та Δb початкове значення нуль
2. Для екземплярів від 1 до m: а. Запустіть процес прямого поширення через усі n_l шарів. Зберігайте вивід активаційної функції у $h^{(l)}$ б. Знайдіть значення $\delta^{(n_l)}$ вихідного шару в. Оновіть $\Delta W^{(l)}$ та $\Delta b^{(l)}$ для кожного шару
3. Запустіть процес градієнтного спуску, використовуючи:

$$W^{(l)} = W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

Значить першим етапом є ініціалізація ваг для кожного шару. Для цього ми використаємо словники у мові програмування Пайтон (позначається через {}). Рандомні значення надаються вагам для того, щоб впевнитись, що нейронна мережа буде працювати правильно під час навчання. Для рандомізації ми використаємо random_sample з бібліотеки numpy. Код виглядає наступним чином:

```
import numpy.random as r
def setup_and_init_weights(nn_structure):
    W = {}
    b = {}
    for l in range(1, len(nn_structure)):
        W[l] = r.random_sample((nn_structure[l], nn_structure[l-1]))
        b[l] = r.random_sample((nn_structure[l],))
    return W, b
```

Наступним кроком є присвоїти двом змінним ΔW та Δb початкові значення нуль (вони повинні мати такий самий розмір, що і матриці ваг та зміщень)Ж

```
def init_tri_values(nn_structure):
    tri_W = {}
    tri_b = {}
    for l in range(1, len(nn_structure)):
        tri_W[l] = np.zeros((nn_structure[l], nn_structure[l-1]))
        tri_b[l] = np.zeros((nn_structure[l],))
    return tri_W, tri_b
```

Далі запустимо процес прямого поширення через нейронну мережу:

```
def feed_forward(x, W, b):
    h = {1: x}
    z = {}
    for l in range(1, len(W) + 1):
        # якщо перший шар, то вагами є x, в іншому випадку,
        # це є виходом з останнього шару
        if l == 1:
            node_in = x
        else:
            node_in = h[l]
        z[l+1] = W[l].dot(node_in) + b[l] # z^(l+1) = W^(l)*h^(l) + b^(l)
        h[l+1] = f(z[l+1]) # h^(l) = f(z^(l))
    return h, z
```

І нарешті, знайдемо вихідний шар $\delta^{(n_l)}$ та значення $\delta^{(l)}$ у прихованих шарах для запуску зворотного поширення:

```
def calculate_out_layer_delta(y, h_out, z_out):
    # delta^(nl) = -(y_i - h_i^(nl)) * f'(z_i^(nl))
    return -(y-h_out) * f_deriv(z_out)


def calculate_hidden_delta(delta_plus_1, w_l, z_l):
    # delta^(l) = (transpose(W^(l)) * delta^(l+1)) * f'(z^(l))
    return np.dot(np.transpose(w_l), delta_plus_1) * f_deriv(z_l)
```

Тепер ми можемо з'єднати усі етапи в одну функцію:

```
def train_nn(nn_structure, X, y, iter_num=3000, alpha=0.25):
    W, b = setup_and_init_weights(nn_structure)
    cnt = 0
    m = len(y)
    avg_cost_func = []
    print('Початок градієнтного спуску для {} ітерацій'.format(iter_num))
    while cnt < iter_num:
        if cnt%1000 == 0:
            print('Ітерація {} від {}'.format(cnt, iter_num))
        tri_W, tri_b = init_tri_values(nn_structure)
        avg_cost = 0
        for i in range(len(y)):
            delta = {}
            # запускає процес прямого поширення та повертає отримані значення h та z, щоб використати у
            # етапі з градієнтним спуском
            h, z = feed_forward(X[i, :], W, b)
            # цикл від nl-1 до 1 зворотного поширення похибок
            for l in range(len(nn_structure), 0, -1):
                if l == len(nn_structure):
                    delta[l] = calculate_out_layer_delta(y[i,:], h[l], z[l])
                    avg_cost += np.linalg.norm((y[i,:]-h[l]))
                else:
                    if l > 1:
                        delta[l] = calculate_hidden_delta(delta[l+1], W[l], z[l])
                    # triW^(l) = triW^(l) + delta^(l+1) * transpose(h^(l))
                    tri_W[l] += np.dot(delta[l+1][:,np.newaxis], np.transpose(h[l][:,np.ne
                    # trib^(l) = trib^(l) + delta^(l+1)
                    tri_b[l] += delta[l+1]
        # запускає градієнтний спуск для ваг у кожному шарі
        for l in range(len(nn_structure) - 1, 0, -1):
            W[l] += -alpha * (1.0/m * tri_W[l])
            b[l] += -alpha * (1.0/m * tri_b[l])
        # завершує розрахунки загальної оцінки
        avg_cost = 1.0/m * avg_cost
        avg_cost_func.append(avg_cost)
        cnt += 1
    return W, b, avg_cost_func
```

Функція зверху має бути трохи пояснена. По-перше, ми не задаємо ліміт роботи градієнтного спуску, базуючись на змінах або точності функції оцінки. Замість цього, ми просто запускаємо його з фіксованим числом ітерацій (3000 у нашому випадку), а потім спостерігаємо, як змінюється загальна функція оцінки з прогресом у навчанні. У кожній ітерації градієнтного спуску, ми перебираємо кожний навчальний екземпляр (range(len(y)) та запускаємо процес прямого поширення, а після нього і зворотне поширення. Етап

зворотного поширення є ітерування через шари, починаючи з вихідного шару до початку - range(len(nn_structure), 0, -1). Ми знаходимо середню оцінку на вихідному шарі (l == len(nn_structure)). Ми також оновлюємо значення ΔW та Δb, позначені як tri_W та tri_b, для кожного шару, крім вихідного (вихідний шар не має жодного зв'язка, який зв'язує його з наступним шаром).

І нарешті, після того, як ми пройшлися по всіх навчальним екземплярам, накопичуючи значення tri_W та tri_b, ми запускаємо градієнтний спуск та змінюємо значення ваг та зміщень:

$$W^{(l)} = W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)} \right]$$

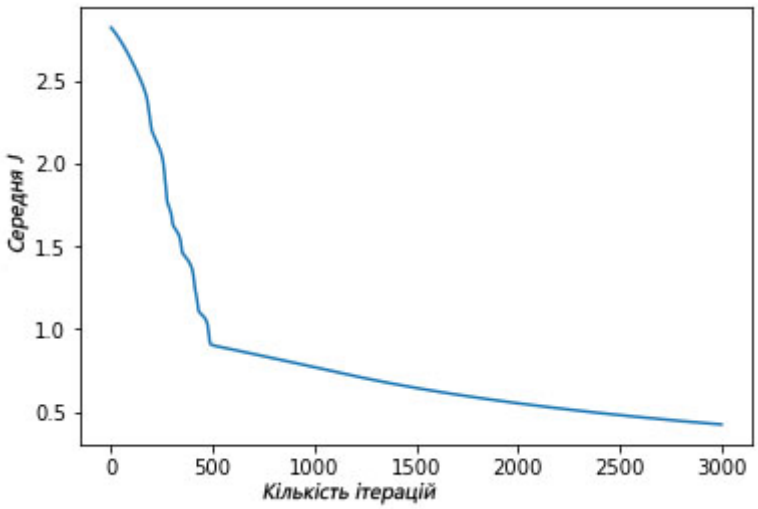
$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

Після закінчення процесу, ми повертаємо отримані вагу та зміщення із середньою оцінкою для кожної ітерації. Тепер час викликати функцію. Її робота може зайняти декілька хвилин, в алежності від вашого комп'ютера.

```
W, b, avg_cost_func = train_nn(nn_structure, X_train, y_v_train)
```

Ми можемо побачити, як функція середньої оцінки зменшилась після ітераційної роботи градієнтного спуску:

```
plt.plot(avg_cost_func)
plt.ylabel('Середня J')
plt.xlabel('Кількість ітерацій')
plt.show()
```



Зверху зображено графік, де зображено, як за 3000 ітерацій нашого градієнтного спуску функція середньої оцінки знизилась і малоімовірно, що подальна ітерація якось змінить результат.

5.5 Оцінка точності моделі

Тепер, після того, як ми навчили нашу нейронну мережу MNIST, ми хочемо побачити, як добре вона працює на тестах. Дано вхідний тест (64 пікселі), нам потрібно отримати вивід нейронної мережі - це робиться через запуск процесу рямого поширення через мережу, використовуючи наші отримані значення ваги та зміщення. Як було сказано раніше, ми обираємо результат вихідного шару через вибір вузла з максимальним виводом. Для цього можна використати функцію numpy.argmax, вона повертає індекс елементу масива з найбільшим значенням:

```
def predict_y(W, b, X, n_layers):
    m = X.shape[0]
    y = np.zeros((m,))
    for i in range(m):
        h, z = feed_forward(X[i, :], W, b)
        y[i] = np.argmax(h[n_layers])
    return y
```

Теперь, нарешті, ми можемо оцінити точність результату (відсоток разів, коли мережа видала правильний результат), використовуючи функцію accuracy_score з бібліотеки scikit learn:

```
from sklearn.metrics import accuracy_score
y_pred = predict_y(W, b, X_test, 3)
```

```
accuracy_score(y_test, y_pred)*100
```

Нам видало результат 86% точності. Звучить досить непогано? Насправді, ні, це є досить поганою точністю. У наш час точність алгоритмів глибинного навчання досягає 99.7%, ми трохи відстали.