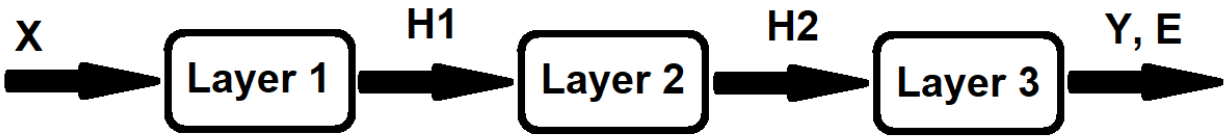
در این مقاله سعی دارم نحوه ی یادگیری شبکه های عصبی چند لایه (MultiLayer Preceptron) و ریاضیات مربوط به این شبکه ها را به صورت کاملا پایه ای توضیح بدم. محوریت این آموزش بر اساس فهم بهتر ریاضیات حاکم بر شبکه های عصبی و توانایی پیاده سازی این محاسبات در زبان های برنامه نویسیه که در این آموزش ما برنامه نویسی پایتون رو جلو خواهیم برد.

تصور من اینه که تا حدودی در مورد شبکه های عصبی میدونید و من سعی دارم نحوه ای استفاده اون رو بیشتر توضیح بدم. خب توضیح بسه بریم سراغ اصل مطلب:

اولا باید در نظر داشته باشیم که شبکه های عصبی رو به شکل لایه لایه در نظر میگیریم و بعد محاسباتمون رو به صورت جداگانه برای هر لایه انجام میدیم. یعنی در هر لحظه فقط یک لایه رو محاسبه میکنیم. منظورمون هم از محاسبه اینه که 1- یه ست داده (یک نمونه از تمام انواع ورودی) رو به لایه اول وارد میکنیم. 2- خروجی رو بر اساس ضرایب (که بهشون وزن هم میگیم) هر کدوم از نرون ها محاسبه میکنیم. 3- خروجی حاصل از لایه ی اول میشه ورودی لایه ی دوم که مرحله ی 2 رو برای لایه ی دوم هم تکرار میکنیم و باز لایه به لایه به همین ترتیب میریم جلو تا به انتهای شبکه ی عصبی برسیم.(Forward Propagation) 4- وقتی به انتهای شبکه ی عصبی یا همون لایه ی خروجی رسیدیم، از تفاوت خروجی که به دست آوردیم با اون خروجی که قرار بود به دست بیاد میزان خطا رو محاسبه میکنیم. (دقت داشته باشیم کی شبکه مون میتونه بیشتر از یک نوع ورودی و خروجی داشته باشه که در این صورت ورودی و خروجی به صورت بردار خواهند بود که عملا تاپیری توی محاسباتمون نداره) 5- از جهت بر عکس که قبلا حرکت میکردیم (یعنی از سمت خروجی به ورودی) مقادیر وزن های نرون ها رو به یه روشی کم و زیاد میکنیم. (Back Propagation) 6- ست داده بعدی رو وارد لایه میکنیم و دوباره همه ی مراحل قبل رو تکرار میکنیم تا تمام داده های موجود یک بار وارد شبکه ی عصبی بشن. 7- این کا رو به دفعات تکرار میکنیم تا میزان خطا رو کمینه کنیم.

خب میدونم که یکم شاید گیج کننده به نظر بیاد ولی در ادامه میخوام تیکه تیکه هر قسمت رو کامل توضیح بدم.

اول شکل زیر رو در نظر داشته باشید:



هر کدوم از لایه های ما از هر نوعی که هستن تو دو تا چیز مشترک هستن و اون ورودی و خروجیه. چیزی که مهمه اینه که خروجی لایه ی قبلی، ورودی لایه ی بعدی به حساب میاد. به عبارتی داده ها از قسمت ورودی به خروجی به صورت لایه به لایه منتشر میشن که به این میگیم انتشار به جلو یا پیش انتشار یا Forward Propagation . وقتی ما در مورد آموزش شبکه ی عصبی داریم صحبت میکنیم یعنی یه سری ورودی داریم که خروجی متناسب با اون ها رو میدونیم و برامون معلوم هستن. شاید با یه مثال بهتر این مساله جا بیوفته. فرض کنید ما از چند تا دکتر کمک گرفتیم تا اطلاعات 1000 نفر رو در مورد سن، قد و وزنشون و این که آیا نسبت به این سه نوع ورودی، وضعیت رشدشون نرمال هست یا آنرمال رو دریافتت کنیم. یعن ما اینجا 1000 نمونه از سه نوع داده ورودی داریم که متناسبا به 1000 نمونه از یک نوع وضعیت خروجی در ارتباطه. یعنی باید سیستم ما سه نوع داده (سن، قد، وزن) از ما بخواد تا بتونه یه نوع داده (رشد نرمال یا آنرمال) رو به ما خروجی بده. یعنی ما از قبل میدونیم که اگر مثلا یه شخصی سنش 26ه، قدش 174ه و وزنش 70 کیلوگرمه رشدش نرمال هست. پس ورودی ها و خروجی های مورد نظر برامون معلوم هستن.

خب برگردیم به بحث. حالا که این سه نوع ورودی برای یک نمونه رو وارد شبکه عصبی مون میکنیم. انتظار داریم خروجی مورد نظرمون رو به ما میده ولی خب میبینیم خروجی شبکه عصبی از خروجی مد نظر ما متفاوته. اختلاف این دو مقدار خروجی میشه خطای شبکه ی عصبی. میزان این خطا رو میشه اینجوری محاسبه کرد:

$$E = \frac{1}{n}\sum_{i}^{n}(\underbrace{y_i^*}_{\text{Target}} - \underbrace{y_i}_{\text{Actuall}})^2$$

هدف ما از آموزش شبکه ی عصبی کمینه کردن این مقداره. اما چطوری؟

بر اساس مقدار این خطا میاییم تغییراتی توی ضرایب هر کدوم از لایه ها میدیم تا مقدار به دست اومده به مقدار مورد نظر نزدیک تر بشه. برای اینکار میتونیم این ضرایب رو به صورت رندوم بالا پایین کنیم و انقدر ادامه بدیم تا خطا کمتر بشه و لی خب به لطف ریاضیات، یه روش هوشمندانه تری برای این کار هست که اسمش گرادیان کاهشیه(Gradient Descent).

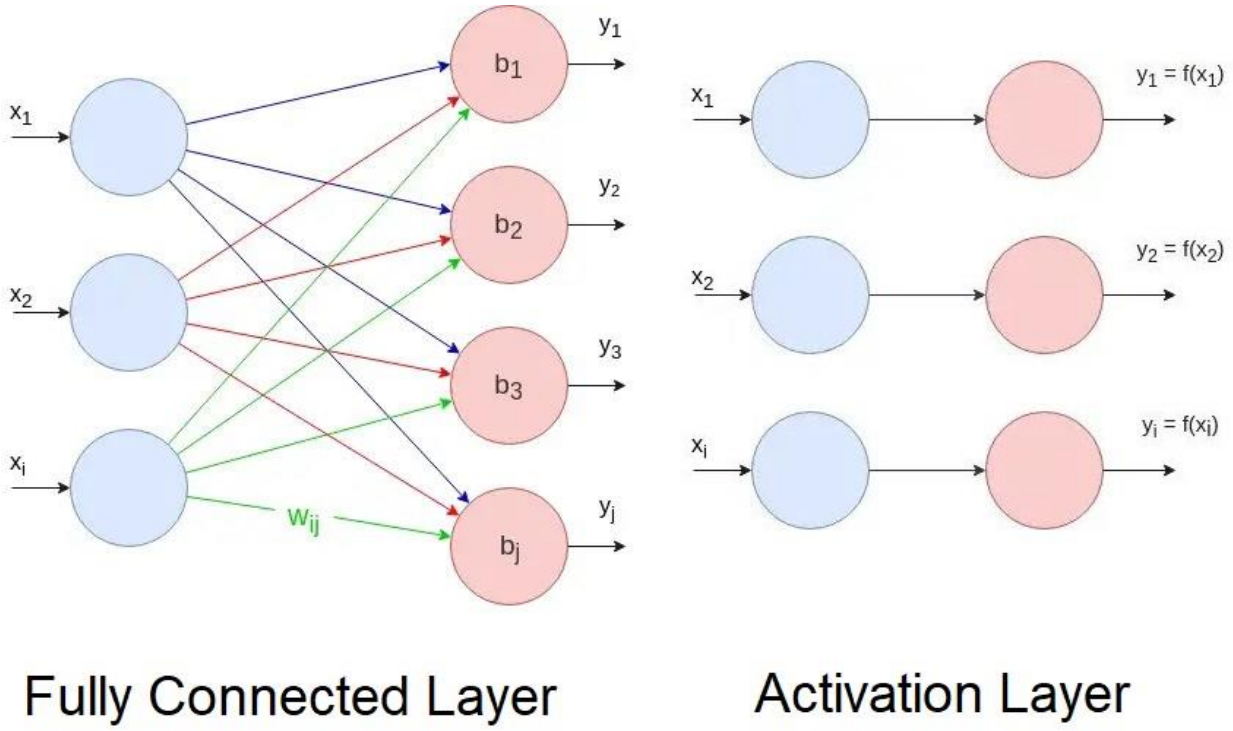
$$w \leftarrow w - \alpha\frac{\partial E}{\partial w}$$

اینجا منظورمون از W همون بردار وزن ها یا ضرایبه که هر نرون روی سیگنالی که بهش وارد میشه اعمال میکنه. فعلا این رو تا اینجا داشته باشید تا ادامه بدیم بعدا تو نحوه ی محاسباتمون برمیگردیم به این قسمت.

همونطور که قبلا هم گفتیم می خوایم تک تک لایه ها رو جداگانه بررسی بکنیم ولی باید دقت داشته باشیم که لایه ها همه یکی نیستن و باید به نسبت تفاوتشون محاسبات متفاوتی براشون داشته باشیم که البته خیلی شبیه به همن این محاسبات. اینجا ما به دو نوع اصلی این لایه ها میپردازیم که یکیش هست لایه ی کاملا متصل (Fully Connected Layer) که در اون همه ی نرون های ورودی به همه ی نرون های

خروجی متصل هستن. خروجی این لایه ها همیشه یه خط رو برای ما میسازه و یه حالت خطی داره ولی خب مگه ما چند تا منحنی داریم که برای مدلسازی اونها یه خط کافی باشه ؟! پس نیاز به یه تابع غیرخطی هم داریم تا بشه هر تابعی رو از خروجی شبکه عصبی درآورد. این تابع رو هم به شکل یه لایه در نظر میگیریم که ورودی و خروجی داره و اسمش رو میذاریم لایه ی فعالسازی و اون تابع رو هم بهش میگیم تابع فعالسازی. این تابع معمولا از توابعی مثل tanh(x) و یا Sigmoid(x) انتخاب میشه.

در ادامه ی شکل یه لایه ی کاملا متصل و لایه فعالسازی رو در یک شبکه عصبی چندلایه ترسیم میکنیم:



خب دیگه توضیحات کلی رو گفتیم و فهمیدیم که شبکه های عصبی چجوری کار میکنن. الان دیگه وقتشه بریم بریم سراغ ریاضیات مساله و بعد هم نحوه ی کد نویسیش. اول با لایه ی کاملا متصل شروع میکنیم. قبلا هم گفتیم که ورودی لزوما از یه نوع داده نیست و تو مثالی هم که آوردیم گفتیم که میشه هر چندتا پارامتر که دوست داشتیم تو ورودی دخیل کنم. این ورودی ها رو میشه به شکل یه بردار یا ماتریس n x i نشون داد که تو اون ما n تا نمونه از i تا پارامتر مختلف داریم. هر کدوم از این ورودی ها یه نرون به

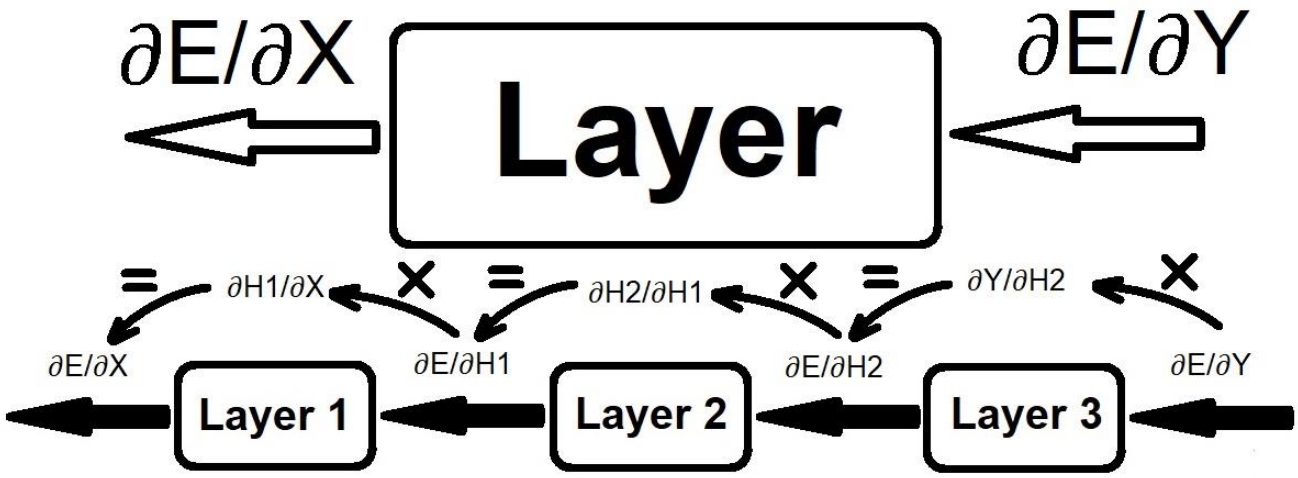
حساب میان. این مقادیر در یه وزن هایی ضرب میشن و با یه مقادیری (بایاس) جمع میشن و خروجی اون لایه رو به دست میدن.میتونیم این محاسبه رو برای یه لایه ای که توش j تا نرون داریم بنویسیم. دقت کنید که هر x به یک نوع داده اشاره داره نه یه نمونه.

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} \dots w_{1j} \\ \vdots \quad \ddots \quad \vdots \\ w_{i1} \dots w_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$

$$Y = XW + B$$

خب قبلا هم گفتیم که رابطه ی بالا انتشار به سمت جلو رو نشون میده. حالا میری برای محاسبه ی انتشار به سمت عقب یا پس انتشار. فرض کنیم اگر ما مشتق خطای خروجی نسبت به خروجی$\frac{\partial E}{\partial Y}$رو به یک لایه بدیم، این شبکه باید به ما مشتق خطای خروجی نسبت به ورودی$\frac{\partial E}{\partial X}$رو تحویل بده. یعنی مثل شکل زیر :

?



یادمون باشه که خطا یا همون E یک عدده و بردار نیست. بنابراین برای X و Y میتونیم اینجوری بنویسیم:

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial x_2} & \cdots & \frac{\partial E}{\partial x_i} \end{bmatrix}$$
$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$

خب با توجه به رابطه ای که برای محاسبه میزان خطا داشتیم میتونیم $\frac{\partial E}{\partial Y}$ رو به سادگی محاسبه کنیم:

$$E = \frac{1}{n}\sum_{i}^{n}(y_i^* - y_i)^2$$
$$\text{Target} \quad \text{Actuall}$$

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_i} \end{bmatrix}$$
$$= \frac{2}{n}\begin{bmatrix} y_1 - y_1^* & \cdots & y_i - y_i^* \end{bmatrix}$$
$$= \frac{2}{n}(Y - Y^*)$$

برای محاسبه ی تغییراتی که باید روی وزن ها و بایاس ها اعمال بشه، نیاز داریم که $\frac{\partial E}{\partial W}$ ، $\frac{\partial E}{\partial B}$ و $\frac{\partial E}{\partial X}$ رو به دست بیاریم. اما چرا $\frac{\partial E}{\partial X}$ ؟ نباید فراموش کنیم که $\frac{\partial E}{\partial X}$ برای یک لایه همون $\frac{\partial E}{\partial Y}$ برای لایه ی قبلی حساب میشه. برای محاسبه این مشتق ها یه قاعده ای داریم به اسم مشتق متوالی یا زنجیره ای که انگار میاییم مشتق یه تابع نسبت به یه متغیر رو میشکنیم و جدا میکنیم و جدا جدا نسبت به یه متغیر دیگه حلش میکنیم.:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

پس برای $\frac{\partial E}{\partial W}$ هم داریم :

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \cdots & \frac{\partial E}{\partial w_{1j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{i1}} & \cdots & \frac{\partial E}{\partial w_{ij}} \end{bmatrix}$$

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_{ij}} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} \\ &= \frac{\partial E}{\partial y_j} x_i \end{aligned}$$

$$\begin{aligned} \frac{\partial E}{\partial W} &= \begin{bmatrix} \frac{\partial E}{\partial y_1} x_1 & \cdots & \frac{\partial E}{\partial y_j} x_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_1} x_i & \cdots & \frac{\partial E}{\partial y_j} x_i \end{bmatrix} \\ &= \begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix} \\ &= X^t \frac{\partial E}{\partial Y} \end{aligned}$$

عالیه! ما تونستیم فرمولی که برای تغییر وزن ها میخواستیم رو به دست بیاریم. بریم سراغ بایاس ها:

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_j} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_j}$$

$$= \frac{\partial E}{\partial y_i}$$

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$= \frac{\partial E}{\partial Y}$$

خب اینم از این. حالا بریم سراغ $\frac{\partial E}{\partial X}$. قبلا هم گفتم اینو برای این محاسبه می کنیم که این مقدار همون $\frac{\partial E}{\partial Y}$ برای لایه ی قبلیشه :

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial x_2} & \cdots & \frac{\partial E}{\partial x_i} \end{bmatrix}$$

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

$$= \frac{\partial E}{\partial y_1}w_{i1} + \ldots + \frac{\partial E}{\partial y_j}w_{ij}$$

$$\frac{\partial E}{\partial X} = \begin{bmatrix} (\frac{\partial E}{\partial y_1}w_{11} + \ldots + \frac{\partial E}{\partial y_j}w_{1j}) & \ldots & (\frac{\partial E}{\partial y_1}w_{i1} + \ldots + \frac{\partial E}{\partial y_j}w_{ij}) \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{i1} \\ \vdots & \ddots & \vdots \\ w_{1j} & \cdots & w_{ij} \end{bmatrix}$$

$$= \frac{\partial E}{\partial Y}W^t$$

خب تموم شد . ما تمام روابطی که برای محاسبه تغییرات مورد نیاز برای تنظیم وزن ها و بایاس های یه لایه ی کاملا متصل لازم داشتیم رو تونستیم به دست بیاریم. یه بار دیگه خلاصه این روابط را مینویسم.

حالا یه بار دیگه این مقادیر رو برای

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^t$$
$$\frac{\partial E}{\partial W} = X^t \frac{\partial E}{\partial Y}$$
$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}$$
$$\frac{\partial E}{\partial Y} = \frac{2}{n}(Y - Y^*)$$

خب الان میریم سراغ لایه تابع فعالسازی ولی خب نترسید، همه ی روابط دقیقا مثل بالاییاست به جز $\frac{\partial E}{\partial X}$ که اونم الان محاسبه میکنیم :

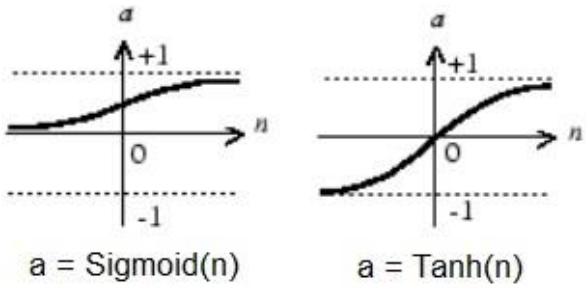
$$\begin{aligned}
\frac{\partial E}{\partial X} &= \begin{bmatrix} \frac{\partial E}{\partial x_1} & \cdots & \frac{\partial E}{\partial x_i} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial x_i} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1} f'(x_1) & \cdots & \frac{\partial E}{\partial y_i} f'(x_i) \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_i} \end{bmatrix} \odot \begin{bmatrix} f'(x_1) & \cdots & f'(x_i) \end{bmatrix} \\
&= \frac{\partial E}{\partial Y} \odot f'(X)
\end{aligned}$$

خب اینم از این. تموم شد ! بریم برای نوشتن کد.

خب این آموزش در واقع آموزش پایتون نیست پس من قصد ندارم پله پله بگم که کد رو چجوری مینویسیم و سعی می کنم جاهایی که احتمال میدم مبهمه توضیح بیشتر بدم. تو کدنویسی پایتون ما یه تعداد توابعی که نیاز داریم ولی تو خود نرم افزار لود نشده رو Import میکنیم. اینجا من توابعی که فکر میکردم نیاز خواهد شد رو اضافه کردم :

```
import sys
import numpy as np                                  # For mathematical calculations
import pandas as pd                                 # To Make DataFrames  For Saving Data
import cryptocompare                                # To Obtain Crypto Data From CryptoCompare API
import csv                                          # To Be Able To Wite in CSV Format
import json                                         # To Work On Object Type Data
from matplotlib import pyplot as plot               # To Plot Graphs
from matplotlib import dates as axis_time           # To Plot Graphs
from datetime import datetime, timedelta            # To Access Datetime
from scipy.signal import savgol_filter              # To Smooth BTC Data Chart
from pandas import Series                           # To Work On Series
import copy                                         # To Save A Copy Of Some Parameters
import warnings                                     # To Ignore The Warnings
warnings.filterwarnings("ignore")
```

خب قبل از ادامه، من تو این قسمت میخوام چند تا تابع که قراره ازشون در ادامه استفاده کنیم رو تعریف عمومی میکنم. یکی از توابعی که تو این زمینه بسیار استفاده میشه تابع تانژانت هایپربولیک و یکی دیگه اش تابع سیگمویده.



به کمک کتابخانه ی پایتون به اسم Numpy می تونیم توابع ریاضی مورد نظر برای لایه ی فعالسازی و همچنین برای محاسبه ی خطا رو اینجا بیاریم.

```
def tanh(x):
    return np.tanh(x)

def tanh_prime(x):
    return 1-np.tanh(x)**2

def sigmoid(x):                          # Sigmoid activation function: f(x) = 1 / (1 + e^(-x))
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(x):                    # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
    fx = sigmoid(x)
   return fx * (1 - fx)

def mse(y_true, y_pred):                 # Loss Function
    return np.mean(np.power(y_true-y_pred, 2));

def mse_prime(y_true, y_pred):           # Derivative Of Loss Function
    return 2*(y_pred-y_true)/y_true.size;
```

خب حالا اول برای هر کدوم از لایه های کاملا متصل و فعالسازی هم دو تا کلاس جدا مینویسیم:

```
class FCLayer(): # Fully Connected Layer    # input_size = number of input neurons

    # output_size = number of output neurons
    def __init__(self, input_size, output_size):
        self.weights = np.random.rand(input_size, output_size) - 0.5
        self.bias = np.random.rand(1, output_size) - 0.5
    # returns output for a given input

    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = np.dot(self.input, self.weights) + self.bias
        return self.output

    # computes dE/dW, dE/dB for a given output_error=dE/dY. Returns input_error=dE/dX.
    def backward_propagation(self, output_error, learning_rate):
        input_error = np.dot(output_error, self.weights.T)
        weights_error = np.dot(self.input.T, output_error)
        # dBias = output_error
        # update parameters
        self.weights -= learning_rate * weights_error
        self.bias -= learning_rate * output_error
        return input_error
```

و کلاس مربوط به لایه ی فعالسازی :

```
class ActivationLayer():
    def __init__(self, activation, activation_prime):
        self.activation = activation
        self.activation_prime = activation_prime

    # returns the activated input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    # Returns input_error=dE/dX for a given output_error=dE/dY.
    # learning_rate is not used because there is no "learnable" parameters.
    def backward_propagation(self, output_error, learning_rate):
        return self.activation_prime(self.input) * output_error
```

خب تقریبا تمام توابعی رو که نیاز داشتیم نوشتیم. الان میتونیم هر موقع به هر کدوم از این نوع لایه ها نیاز داشتیم فقط با فراخوانی اونها کار رو جلو ببریم. حالا وقت اینه که به کمک یه کلاس دیگه که من اسمشو Network گذاشتم همه ی این قسمت ها رو جمع بندی کنیم:

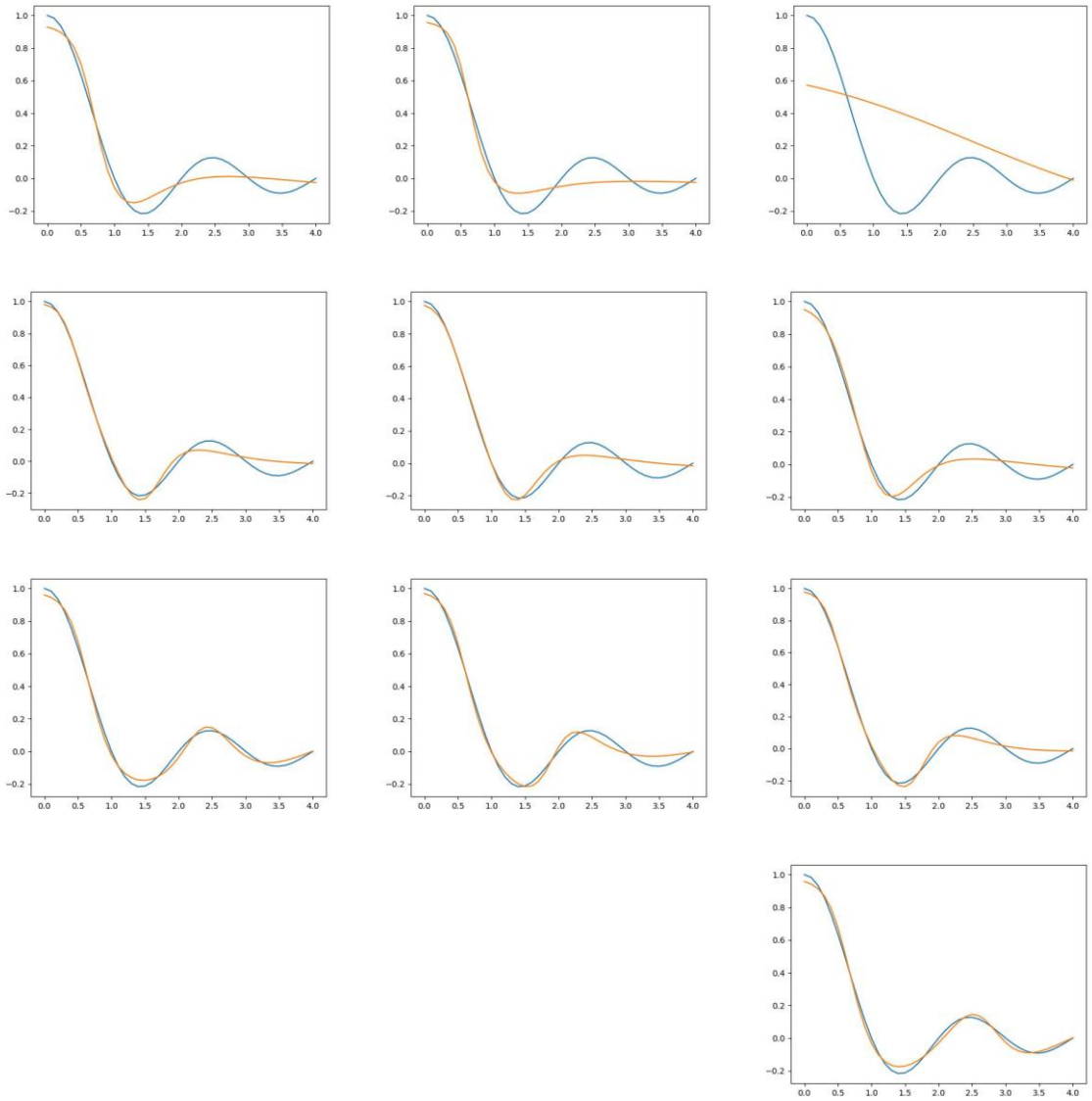
```
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.loss_prime = None

    # add layer to network
    def add(self, layer):
        self.layers.append(layer)

    # set loss to use
    def use(self, loss, loss_prime):
        self.loss = loss
        self.loss_prime = loss_prime

    # predict output for given input
    def predict(self, input_data):
        # sample dimension first
        samples = len(input_data)
        result = []
        # run network over all samples
        for i in range(samples):
            # forward propagation
            output = input_data[i]
            for layer in self.layers:
                output = layer.forward_propagation(output)
            result.append(output)
        return result
```

```
# train the network
def train(self, x_train, y_train, epochs, learning_rate):
    # sample dimension first
    samples = len(x_train)
    # training loop
    for i in range(epochs):
        err = 0
        for j in range(samples):
            # forward propagation
            output = x_train[j]
            for layer in self.layers:
                output = layer.forward_propagation(output)
            # compute loss (for display purpose only)
            err += self.loss(y_train[j], output)

            # backward propagation
            error = self.loss_prime(y_train[j], output)
            for layer in reversed(self.layers):
                error = layer.backward_propagation(error, learning_rate)
        # calculate average error on all samples
        err /= samples
        if i % 500 == 0 :
            print('epoch %d/%d   error=%f' % (i+1, epochs, err))
            #plot.figure()
            #plot.plot(x_train,np.squeeze(np.array(self.predict(x_train))))
            #plot.show(block=False)
```

خب الان وقت چیه ؟ بعلع وقت بازی با چیزی که خلق کردیم. میخوام یه شبکه عصبی سه لایه که یه ورودی داره، پنج تا نورون تو لایه ی اول، چهارتا نورون تو لایه ی دوم و یه خروجی داره رو بسازیم و یه تابع به عنوان ورودی به شبکه عصبیم بدم و ازش بخوام خودش رو جوری تنظیم کنه که هر موقع من ورودی بهش دادم یه نمونه از اون تابع بسازه. به این کار میگن Curve Fitting. تابعی که بهش میدم تابع Sinc(x) ه بین بازه ی صفر تا چهار:

```
x = np.linspace(0, 4, 41)
y = np.sinc(x)
x_train = x
y_train = y
net = Network()
net.add(FCLayer(1, 5))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(5, 4))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(4, 1))
net.add(ActivationLayer(tanh, tanh_prime))
# train
net.use(mse, mse_prime)
net.train(x_train, y_train, epochs=5000, learning_rate=0.05)

# test
#out = net.predict(x_train[:,0,0])
out = net.predict(x)
out = np.squeeze(np.array(out))
plot.plot(x,y)
plot.show(block=False)
plot.plot(x,out)
plot.show(block=False)
```

در ادامه شکل هارو رسم میکنم که ببینیم تو هر پونصد پله که محاسبات رو تکرار میکنه چه اتفاقی میوفته.



خب چی بهتر از این ؟! میبینیم که شبکه عصبی داره نسبت به ورودی هایی که بهش دادیم آموزش دیده و با این که هیچ تصوری از تابع Sinc(x) نداره ولی با آموزشی که دیده میتونه تا حدودی شبیه به اون رفتار داشته باشه.

تو این مقاله سعی شده تا همه چیز ساده باشه. امیدوارم تونسته باشم کمکتون کنم. ممنون که تا اینجا خوندین.