به نام خدا

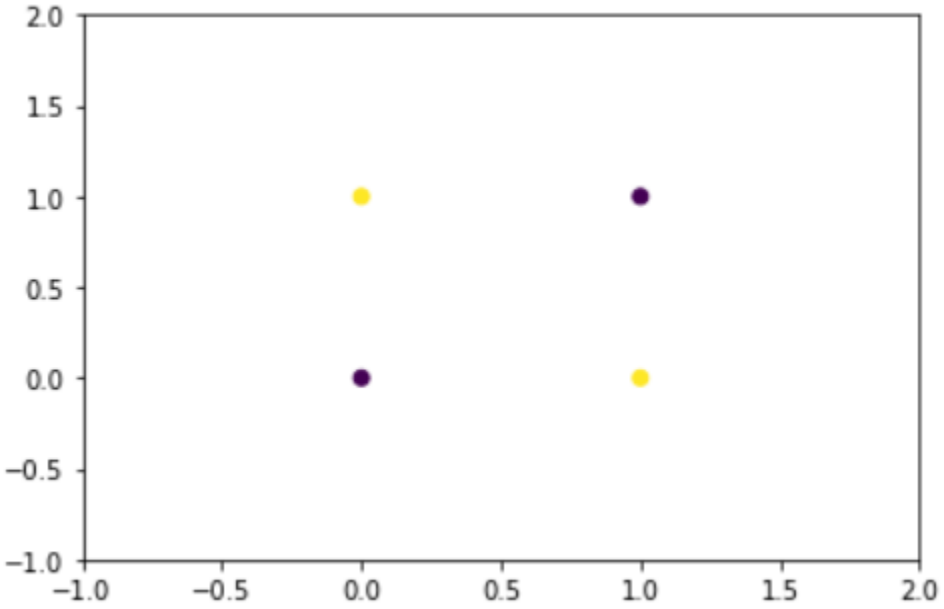
تمرین یک درس شبکه های عصبی توحید عابدینی ۹۹۷۲۲۱۵۳

سوال ۱)

بخش a)

به دلیل اینکه داده های ما قابل تفکیک پذیر خطی نیستند نمیشود با یک نورون عصبی آنها را از هم جدا سازی کرد. در هر حالتی یکی از داده ها اشتباه کلاس بندی میشود.

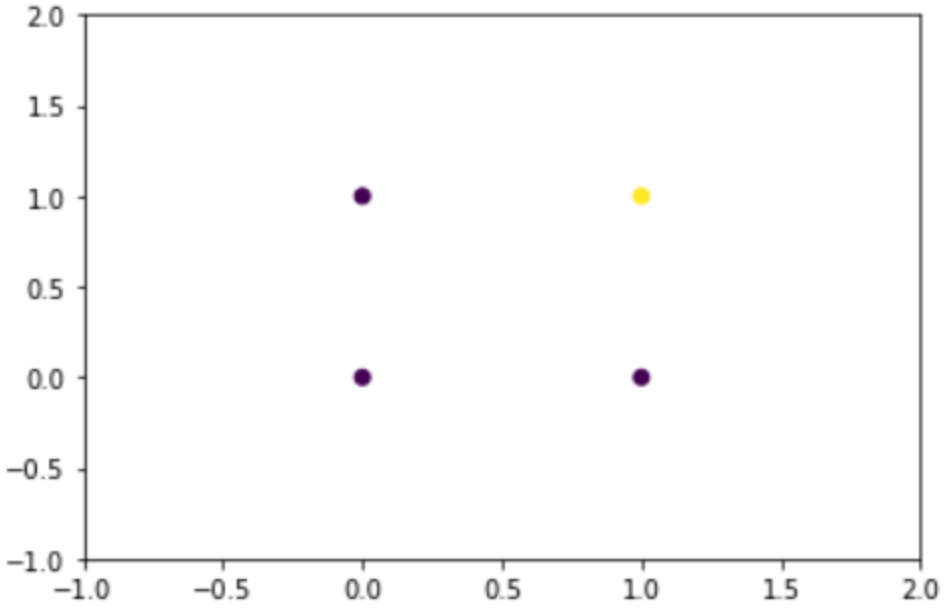
نمودار داده ها به شکل زیر است. که در آن رنگ بنفش بیانگیر مقدار یک و رنگ زرد بیانگر مقدار صفر میباشد.



بخش b)

داده های داده شده به صورت خطی قابل تفکیک هستند و بیانگر تابع NAND منطقی میباشند.

نمودار داده ها به شکل زیر است. که در آن رنگ بنفش بیانگیر مقدار یک و رنگ زرد بیانگر مقدار صفر میباشد.



میبینم که این داده ها با یک خط قابل تفکیک هستند و شروع به محاسبه وزن ها میکنیم.
وقتی که هر دو مقدار ورودی ما یک باشند تابع مقدار 0 را برمیگرداند. و در بقیه حالات مقدار 1 را برمیگرداند.
یک راه حل پیشنهادی داشتن تعدادی وزن منفی است. مثلا برای هر دو وزن w_1 و w_2 مقدار 0.5- را بدهیم.
حال با محاسبه روی تک تک نقاط شروع به یافتن مقدار b (Threshold) مناسب میکنیم.
در داده اول که 0,0 است و خروجی 1 است داریم:

$$y = 0 * (-0.5) + 0 * (-0.5) + b = b$$

در داده دوم که 0,1 است و خروجی 1 است داریم:

$$y = 0 * (-0.5) + 1 * (-0.5) + b = b - 0.5$$

در داده سوم که 1,0 است و خروجی 1 است داریم:

$$y = 1 * (-0.5) + 0 * (-0.5) + b = b - 0.5$$

در داده چهارم که 1,1 است و خروجی 0 است داریم:

$$y = 1 * (-0.5) + 1 * (-0.5) + b = b - 1$$

حال مقدار b باید به گونه ای باشد که با تابع فعال ساز زیر خروجی شبکه را در هر 4 حالت صحیح برگرداند.

تابع فعال ساز:

$$out = \begin{cases} 1 \text{ if } y >= 0 \\ 0 \text{ if } y < 0 \end{cases}$$

یعنی b و 0.5 – b مقداری مثبت بوده و b-1 مقداری منفی شود.
یک مقدار مناسب برای b مقدار 0.7- است.
پس مقادیر ما عبارت اند از:

w_1= -0.5, w_2= -0.5, b=-0.7

سوال ۲)

نحوه عملکرد یک نورون OR به شرح زیر است:

x1	x2	y
-1	-1	-1
-1	1	1
1	-1	1
1	1	1

برای طراحی یک نورون OR باید ابتدا مقادیر اولیه وزن ها و بایاس را مشخص کنیم و سپس شروع به آپدیت کنیم.
مقادیر پیشنهادی برای شروع به صورت زیر است:

$w_1=w_2=0.3$ و $b=0$

$\eta=0.1$

مقدار نه چندان زیاد و نه چندان کم برای نرخ یادگیری (η) پیشنهاد شده است (مشابه اسلاید ها). میدانیم مقدار زیاد باعث overshooting شده و مقادیر کم هم سرعت همگرایی را به شدت کند میکنند.
مقادیر وزن ها هم مقادیر کوچک و بین 1و1- انتخاب شده است.

از روابط زیر برای به روز رسانی استفاده میکنیم:

Neuron input	Delta rule
$y = b + x_i w_i$	$b(new) = b(old) + \eta(d - y)$ $w_i(new) = w_i(old) + \eta(d - y)x_i$

گام های به روز رسانی وزن ها)

1)

برای ورودی 1- , 1- که خروجی 1- دارد.

$y = 0.3 * (-1) + 0.3 * (-1) + 0 = -0.6$
$w_1 = 0.3 + 0.1 (-1 – (-0.6)) = 0.26$
$w_2 = 0.3 + 0.1 (-1 – (-0.6)) = 0.26$
$b = 0 + 0.1 (-1 – (-0.6)) = - 0.04$

2)

برای ورودی 1 , 1- که خروجی 1 دارد.

$y = 0.26 * (1) + 0.26 * (-1) + (-0.04) = -0.04$
$w_1 = 0.26 + 0.1 (1 – (-0.04)) = 0.364$
$w_2 = 0.26 + 0.1 (1 – (-0.04)) = 0.364$
$b = - 0.04 + 0.1 (1 – (-0.04)) = 0.064$

3)

برای ورودی 1- , 1 که خروجی 1 دارد.

$y = 0.364* (-1) + 0.364* (1) + 0.064 = 0.064$
$w_1 = 0.364 + 0.1 (1 – 0.064) = 0.4576$
$w_2 = 0.364 + 0.1 (1 – 0.064) = 0.4576$
$b = 0.064 + 0.1 (1 – 0.064) = 0.1576$

4)

برای ورودی 1 , 1 که خروجی 1 دارد.

$y = 0.4576* (1) + 0.4576* (1) + 0.1576 = 1.0728$
$w_1 = 0.4576+ 0.1 (1 – 1.0728) = 0.45032$
$w_2 = 0.4576+ 0.1 (1 – 1.0728) = 0.45032$
$b = 0.1576+ 0.1 (1 – 1.0728) = 0.15032$

پس از 4 بار به روز رسانی وزن ها در نهایت به مقادیر زیر میرسند.

$w_1 = 0.45032$
$w_2 = 0.45032$
$b = 0.15032$

y = -1 * (0.45032) + (-1) * (0.45032) + 0.15032 = -0.75032 => -1
y = 1 * (0.45032) + (-1) * (0.45032) + 0.15032 = 0.15032=> 1
y = -1 * (0.45032) + 1 * (0.45032) + 0.15032 = 0.15032=> 1
y = 1 * (0.45032) + 1 * (0.45032) + 0.15032 = 1.05096=> 1

که میبینیم برای هر چهار حالت ورودی با تابع فعال ساز زیر درست عمل میکند و به اصطلاح شبکه همگرا شده است.

$$out = \begin{cases} 1 \text{ if } y >= 0 \\ -1 \text{ if } y < 0 \end{cases}$$

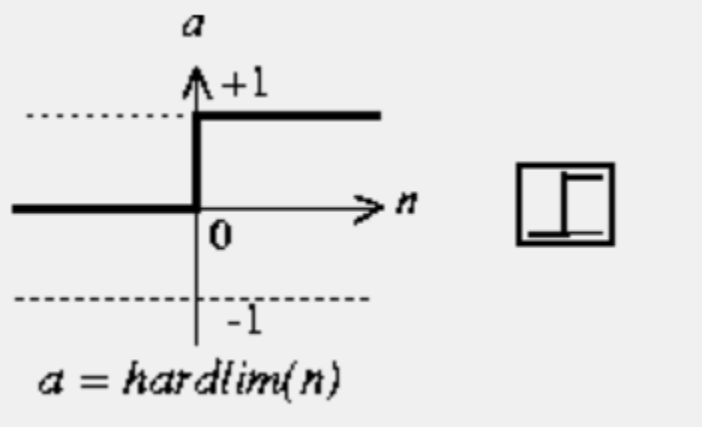
سوال ۳)

بخش a)

پرسپترون محدودیت های متعددی دارد که به آنها اشاره میکنیم:

خروجی پرسپترون به دلیل محدودیت های تابع hard-limit transfer فقط مقادیر ۰ و ۱ را میتواند داشته باشد.

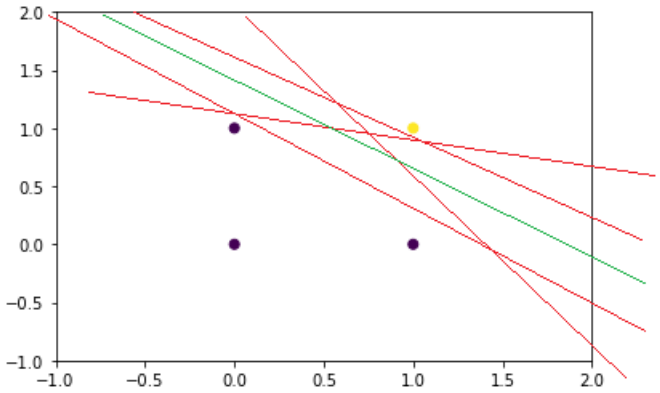
در شکل زیر تابع مذکور را مشاهده میکنیم:



پرسپترون فقط قادر به تفکیک ناحیه ها یا داده های خطی تفکیک پذیر است که متاسفانه در دنیای واقع بخش خیلی کمی از داده های ما اینگونه هستند (در حدی که عملا میتوانیم بگوییم نیستند.) و این کارایی عملی و واقعی پرسپترون را زیر سوال میبرد. البته شایان ذکر است که برای همین مسائل خطی تفکیک پذیر پرسپترون در نهایت مساله را در زمان محدود میتواند حل کند.

مساله ی دیگری که به عنوان یکی از معایب پرسپترون مطرح میشود این است که پرسپترون رسیدن به جواب را برای آن مجموعه داده تضمین میکند (در زمان محدود) اما راجع به اینکه ناحیه مورد نظر به طور مناسب و با

فاصله های مساوی از نواحی مرزی دو کلاس قرار دارد تضمینی ندارد. به عنوان مثال در شکل زیر تمامی جواب ها قابل قبول است اما بدیهی است خطی که margin مناسب تری داشته باشد و فاصله مساوی داشته باشد جواب بهتری است (به صورت نسبی میتوان گفت خط سبز از بقیه خطوط قرمز مارجین بهتری دارد.) و پرسپترون در این باره تضمینی ندارد.



منبع : http://matlab.izmiran.ru/help/toolbox/nnet/percep11.html

بخش b)

تابع sigmoid:

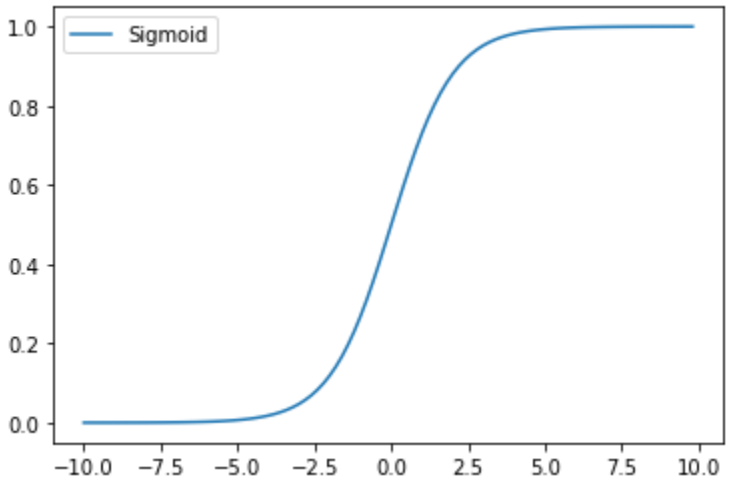
ضابطه تابع :

$$S(x) = \frac{1}{1 + e^{-x}}$$

$S(x)$ = sigmoid function

e = Euler's number

نمودار تابع:



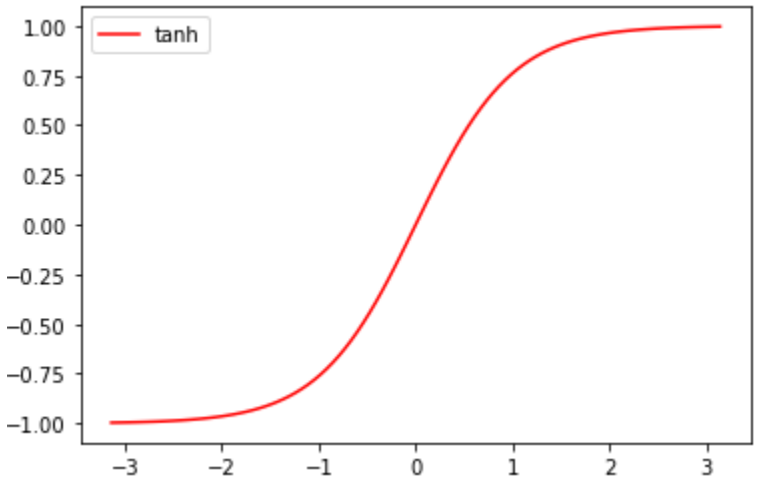
تابع sigmoid غیرخطی است و خروجی آنالوگ میدهد. (بر خلاف تابع پله ای). گرادیان هموارى دارد و برای یک طبقه بند مناسب است. خروجی تابع فعال ساز همیشه بین 0 و 1 بوده که در مقایسه با توابع خطی و یا ReLU که رنج نامحدود دارند محدود تر است و به همین دلیل فعال سازی شدیدی نخواهد داشت و کنترل شده است. در طول طرف تابع واکنش به تغییرات خیلی کم بوده و تقریبا هموار است و مشکل ناپدید شدن گرادیان را داریم. خروجی اش با مرکز صفر نیست (مرکزش مقدار 0.5 است) و بروزرسانی ها بر اساس گرادیان خیلی کم در دو جهت متفاوت خواهند شد. این تابع اشباع میشود و کارآیی اش را در مقادیر بالا از دست میدهد. همگرایی کندی دارد و محاسبات نسبتا هزینه بری هم دارد.

تابع tanh:

ضابطه تابع:

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

نمودار تابع:



این تابع هم همانند sigmoid غیر خطی بوده و گرادیان نسبتا هموار دارد (البته به نسبت sigmoid کمی شیب بیشتری دارد.) خروجی تابع فعال ساز همیشه بین 1- و 1 بوده که در مقایسه با توابع خطی و یا ReLU که رنج نامحدود دارند محدود تر است و به همین دلیل فعال سازی شدیدی نخواهد داشت و کنترل شده است. در طول طرف تابع واکنش به تغییرات خیلی کم بوده و تقریبا هموار است و مشکل ناپدید شدن گرادیان را داریم. این تابع اشباع میشود و کارآیی اش را در مقادیر بالا از دست میدهد. همگرایی کندی دارد و محاسبات نسبتا هزینه بری هم دارد.

تابع softmax:

ضابطه:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ = softmax

$\vec{z}$ = input vector

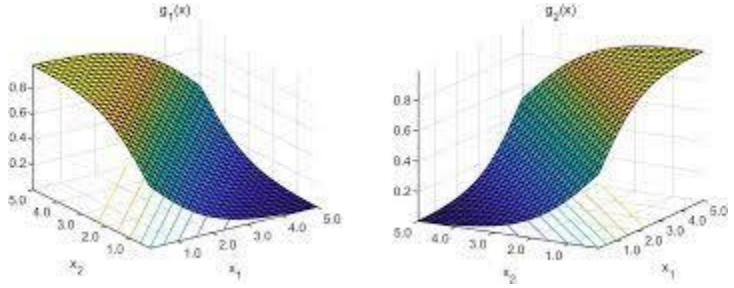
e^{z_i} = standard exponential function for input vector

K = number of classes in the multi-class classifier

e^{z_j} = standard exponential function for output vector

e^{z_j} = standard exponential function for output vector

نمودار:
به دلیل ماهیت تابع softmax و چند متغیره بودن آن امکان رسم در فضای دو بعدی وجود ندارد.
برای حالت دو کلاسه نمودار زیر را میتوان نشان داد:



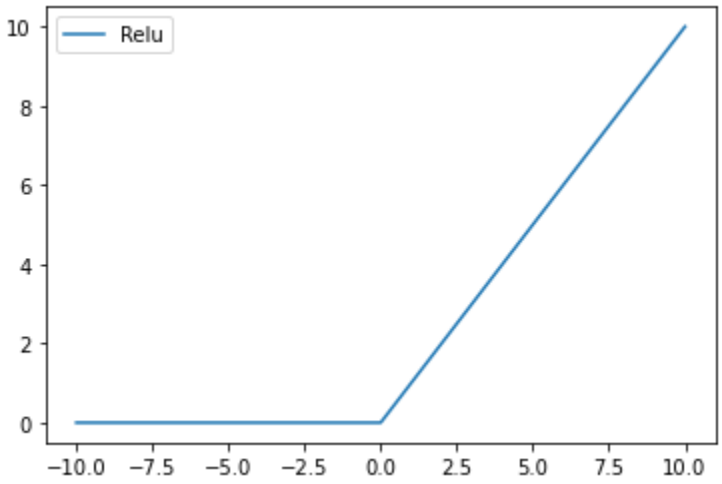
تابع softmax احتمالات را محاسبه میکند و برای هر کلاس از کلاس های ما یک احتمال رخداد محاسبه میکند که مشخصا مجموع آنها باید 1 شود. میتوان گفت از ترکیب چند sigmoid ساخته میشود. یکی از مناسب ترین توابع برای انجام کلاس بندی چند کلاسه است. از این تابع بیشتر برای نورون های لایه آخر (خروجی) استفاده میشود.

تابع ReLU:
ضابطه:

$$ReLU$$

$$f(x) = max\,(0, x)$$

نمودار:



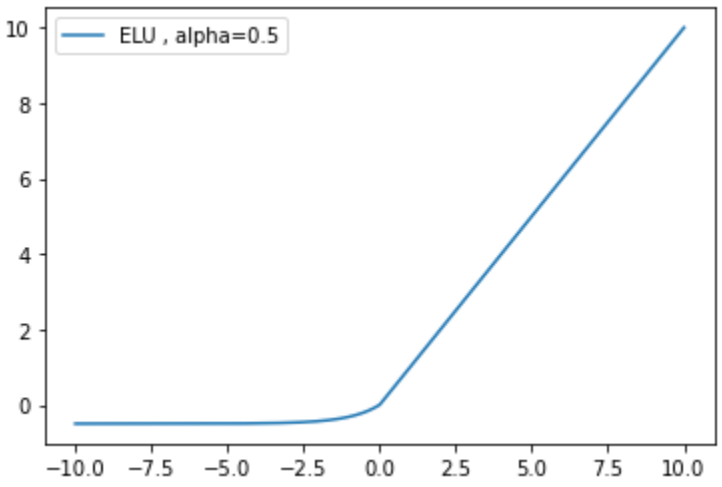
تابع ReLU بسیار ساده تر از توابع قبل است و محاسبات هزینه بری ندارد و مشکل ناپدید شدن گرادیان را هم حل میکند. فقط قابل استفاده در لایه های میانی شبکه است. برخی گرادیان ها (برای مقادیر منفی که همواره صفر هستند) میتوانند شکننده باشند و یک نورون را به کلی از کار بیندازند و به اصطلاح نورون بمیرد. این تابع از سمت مثبت ها محدود نیست و امکان فعال سازی بسیار شدید را دارد. تابع ReLU در صفر مشتق پذیر نیست. سرعت همگرایی این تابع بیشتر است.

تابع ELU:

ضابطه:

$$R(z) = \begin{cases} z & z > 0 \\ \alpha.(e^z - 1) & z <= 0 \end{cases}$$

نمودار:



یک تغییر اعمال شده روی مقادیر منفی تابع ReLU منجر به ایجاد تابع ELU میشود.یک جایگزین قدرتمند برای ReLU است که مشکلات نورون های مرده را ندارد و همچنین میتواند مقادیر منفی هم تولید کند. اما همچنان برای مقادیر مثبت امکان انفجار گرادیان و فعال سازی شدید را دارد. سرعت همگرایی این تابع بیشتر است.

منابع:

https://tcoil.info/activation-functions-sigmoid-tanh-relu
https://www.v7labs.com/blog/neural-networks-activation-functions
https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html
https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-706ac4284c8a

بخش c)

معایب SGD:

زمان طولانی تری برای اتمام یک ایپاک نسبت به GD نیاز دارد.

زمان بیشتری برای همگرا شدن نیاز دارد.

احتمال گیر کردن در مینیمم محلی و نرسیدن به مینیمم سراسری را دارد.

توضیح SGD + momentum:

میدانیم در روش Mini Batch SGD آپدیت وزن های نویزی است. با کمک اضافه کردن momentum به denoise کردن گرادیان ها میپردازیم. ایده اصلی برای این رفع نویز استفاده از میانگین وزن دار نمایی است که ضریب بیشتری به آپدیت های اخیر (قدیمی) داده و اثر بالای آپدیت قبلی را کاهش میدهد.

قوانین آپدیت وزن ها و ترم سرعت (ممان یا momentum) به صورت زیر است:

$$W_{new} = W_{old} - lr * (\nabla_W L)_{W_{old}}$$

$$V_{new} = \alpha * V_{old} + lr * (\nabla_W L)_{W_{old}}$$

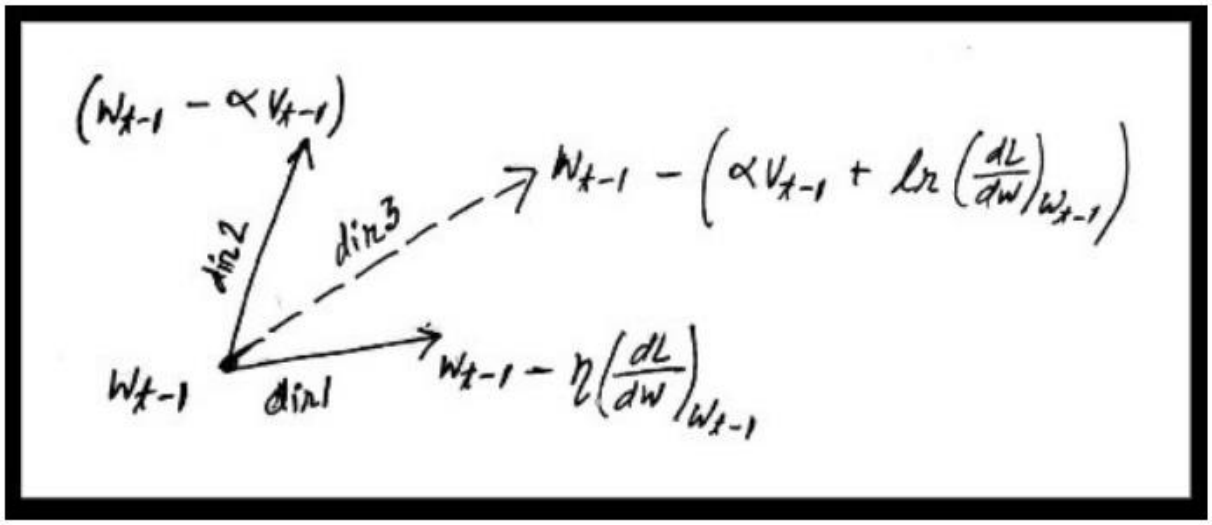
$$W_{new} = W_{old} - (\alpha * V_{old} + lr * (\nabla_W L)_{W_{old}})$$

$$0 < \alpha < 1$$

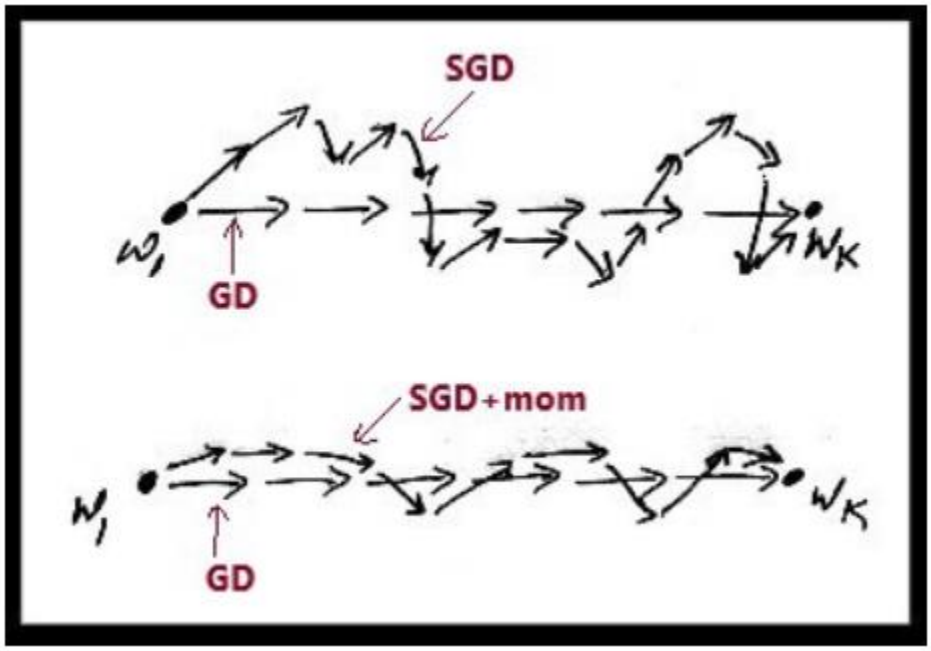
momentum در زمان t به کمک وزن دهی به تمامی آپدیت های اخیر در مقایسه با آپدیت قبلی محاسبه میشود که به سریع تر شدن همگرایی کمک میکند. V_{old} همان بخش موثر از آپدیت های اخیر است که به کمک ما می‌آید.

حال کمی جزئی تر به اینکه چگونه به سرعت بیشتر میرسیم نگاه میکنیم.

اگر برای آپدیت وزن ها فقط از گرادیان استفاده کنیم تغییر در جهت dir1 خواهد بود. اگر فقط از momentum برای آپدیت استفاده کنیم تغییرات ما در جهت dir2 خواهد بود و اما اگر از هر دو عبارت برای آپدیت استفاده کنیم تغییرات ما در جهت dir3 خواهد بود که این رفع نویز ذکر شده را هم انجام خواهد داد. ابتدا در شکل زیر آنچه که ذکر شد را ببینیم:



حال مقایسه ای در نحوه حرکت بین الگوریتم های SGD و GD و momentum خواهیم داشت.



مشاهده میکنیم که با استفاده از momentum حرکت های ما کمی هدفمند تر شده است و نویز ندارد و در نتیجه سریع تر به همگرایی میرسیم.

مقدار مناسب آلفا هم طبق تجربه مقدار بالایی نظیر 0.9 یا 0.95 است.

مزایای این روش شامل تمامی مزایای SGD هست و دارای همگرایی سریع تر از GD هم هست.

اما به عنوان معایب میتوان به محاسبت بیشتر برای هر بار بروز رسانی اشاره کرد.

توضیح RMS prop :

RMS prop یا Root Mean Square Propogation یکی دیگر از الگوریتم های بهینه سازی است که به توضیح آن میپردازیم.

به روابط زیر توجه کرده و به ترتیب به توضیح آنها میپردازیم:

$$For\ each\ Parameter\ w^j$$

$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \rho\nu_{t-1} + (1-\rho) * g_t^2$$

$$\Delta\omega_t = -\frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$$\eta : Initial\ Learning\ rate$$
$$\nu_t : Exponential\ Average\ of\ squares\ of\ gradients$$
$$g_t : Gradient\ at\ time\ t\ along\ \omega^j$$

در رابطه اول همانند آنچه که در momentum گفته شد نیاز به میانگین وزن دار نمایی داریم. و دلایل مربوطه در همان قسمت ذکر شده است. اما نحوه به روز رسانی ما کمی فرق دارد.

در رابطه دوم مقدار گام را تعیین میکنیم. باید در جهت گرادیان حرکت کنیم اما مقدار نرخ یادگیری بر رادیکال میانگین به دست آمده از رابطه قبل تقسیم میشود.

رابطه سوم هم فقط شامل به روز رسانی است. هایپرپارامتر ρ معمولا مقدار بالا (0.9) دارد.

در مخرج یک اپسیلون هم گذاشته شده است تا از صفر شدن مخرج جلوگیری کند.

مزایا:

نرخ یادگیری بر مبنای گرادیان های آخیر تغییر میکند و به اصطلاح تطبیق پیدا میکند. بدین صورت که در موقع لزوم برای کاهش برای جلو گیری از انفجار و فعال سازی شدید اقدام به کنترل کرده و در مواقعی که مقدار نرخ یادگیری خیلی پایین است اقدام به افزایش آن میکند تا در مینمم محلی گیر نکنیم.

در این الگوریتم هر هایپر پارامتر نرخ یادگیری مخصوص خود را دارد. به طور کلی نرخ یادگیری به طور **بهینه** رفتار میکند و بدون دلیل کاهش ندارد.

معایب:

سابقه میانگین های گرادیان های قبلی را همانند momentum نگه نمیدارد.

منبع:

https://towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5
https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam
https://medium.com/nerd-for-tech/optimizers-for-neural-networks-a74cb4152307
https://medium.com/swlh/strengths-and-weaknesses-of-optimization-algorithms-used-for-machine-learning-58926b1d69dd