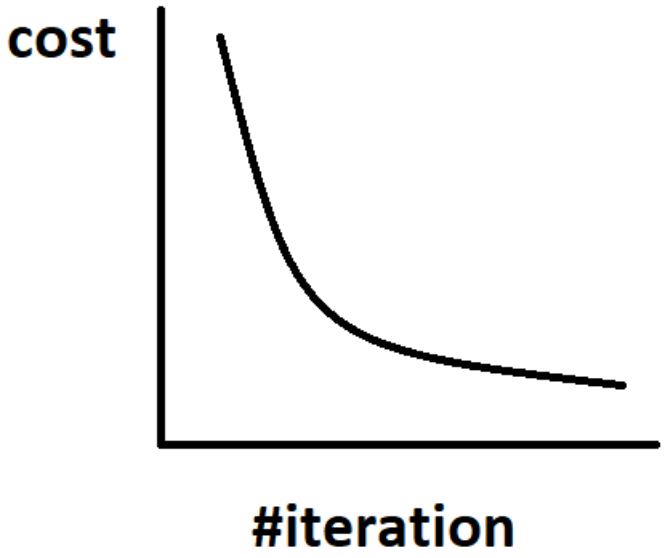
۱-بخش اول: تفاوت batch، mini batch و stochastic:

روش gradient descent به این صورت است که ما برای پیدا کردن بهترین پارامترهای wوb برای شبکه ی عصبی ای که درنظرگرفتیم یک مرحله ی فوروارد در نظرمیگیریم و با استفاده از فرمول خطی z=wx+b و استفاده از توابع فعال سازی در هر لایه در صورت نیاز در نهایت به خروجی لایه ی آخر میرسیم.در اینجا یک تابع هزینه تعریف میکنیم که به نوعی وابسته به نوع مسئله خروجی حاصل از مرحله ی فوروارد را با خروجی اصلی که توقع داریم داشته باشیم مقایسه میکند و تغییراتی انجام میدهد. خواسته ی ما این است که این تابع هزینه که تابعی از این اختلاف مقدار خروجی واقعی و خروجی بدست آمده است رفته رفته کاهش پیدا کند. سپس وارد فاز بعدی الگوریتم یعنی انتشار بک وارد میشویم. در این فاز ما از خروجی هر لایه مشتق میگیریم و با معادلات ریاضی در هر مرحله بخشی از ارور را به لایه ی قبلی انتقال میدهیم و وزن ها و ترشلدهای هر لایه را به روز رسانی میکنیم. سپس دوباره با وزن های جدید مرحله ی فوروارد انجام میگیرد و تابع هزینه محاسبه میشود. با این عملیات همانطور که انتظار داریم در هر بار اجرای الگوریتم تابع هزینه کاهش پیدا میکند مانند شکل زیر:



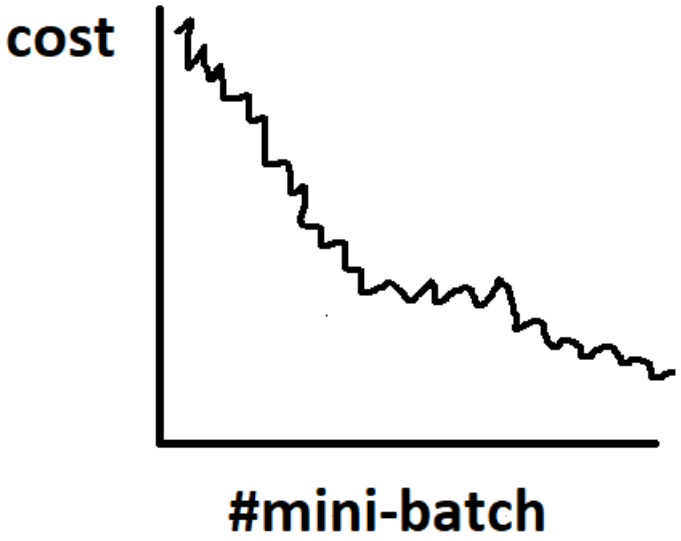
به طور کلی میتوانیم بگوییم این روش گرادیان نزولی به دو شکل اجرا میشود.

۱-batch gradient descent

۲-mini-batch gradient descent

در اولی وقتی که تعداد مثال های آموزشی زیاد باشد مثلا چند میلیون نمونه ی آموزشی داشته باشیم بعد از یک قدم کوچک و یک به روزرسانی کوچک روی وزن ها دوباره باید الگوریتم روی کل نمونه ها اجرا شود و هزینه ی زمانی و سخت افزاری زیادی دارد برای همین روش دوم هم پیشنهاد شد که بخشی از داده ها الگوریتم را پیش ببرند تا سریعتر پیش برویم.

در اولی در هر مرحله cost مرتبا کاهش پیدا میکند اما در mini-batch ممکنه در هر تکرار cost کاهش پیدا نکنه چون به هرحال در هر epoch روی داده های جدیدی train میکنید. به طور کلی نمودار cost این حالت هم نزولی است اما با نویز همراه است در نتیجه اینکه در هربار مشتق گرفتن نسبت به قبل کاهش نیابد طبیعی است. ن.سان نمودار به خاطر این است که یک دسته داده ممکن است خیلی آموزش آسان تری داشته باشد و سریع تر هزینه در آن کاهش پیدا کند اما یک دسته آموزش سخت تری داشته باشد.



اگر mini-batch-size=n باشد این روش تبدیل میشود به روش batch فقط یک دسته داده داریم و در هر epoch الگوریتم روی کل همان یک دسته اجرا میشود.

اگر mini-batch-size=1 باشد، این روش تبدیل میشود به الگوریتم stochastic که هر نمونه ی آموزشی یک دسته محسوب میشود و در هر epoch الگوریتم روی یک نمونه اجرا خواهد شد.

در حالت batch الگوریتم خیلی طول میشه در هر epoch و همینطور موجب از دست دادن speed-up در vectorization میشه.

در حالت stochastic چون فقط با یک نمونه گرادیان بررسی میشه در بیشتر مواقع دو عدد به عنوان مینیمم گلوبال ارائه خواهد شد و گاهی اوقات نقطه ی اشتباه برای ادامه ی مسیرانتخاب خواهد شد در نتیجه خیلی نویز دارد.

یک epoch معمولاً به این معنی است که الگوریتم شما هر نمونه آموزشی را یک بار می بیند به عبارتی یک حلقه در کل مجموعه داده است. حال با فرض اینکه n نمونه آموزشی دارید:

اگر به‌روزرسانی دسته‌ای(batch) را اجرا می‌کنید، برای هر بار به روز رسانی پارامترهای مسءله لازم است الگوریتم شما رو ی تمام n نمونه ی آموزشی اجرا شود، یعنی در هر epoch، پارامترهای شما یک بار به‌روزرسانی می‌شوند.

اگر به‌روزرسانی مینی دسته‌ای را با اندازه دسته = b اجرا می‌کنید، به ازای هربار به روز رسانی شدن پارامترهای مسئله الگوریتم شما نیاز دارد که روی b نمونه از n نمونه آموزشی اجرا شود، یعنی در هر epoch، پارامترهای شما حدود n/b بار به‌روزرسانی می‌شوند.

اگر به‌روزرسانی SGD که مخف stochastic gradient discentاست را اجرا می‌کنید، هر بار به‌روزرسانی پارامتر نیاز دارد الگوریتم شما را روی 1 مورد از n نمونه آموزشی را ببینید، یعنی در هر epoch، پارامترهای شما حدود n بار به‌روزرسانی می‌شوند. این روش در واقع نوعی خاص از mini-batch gradient descent است که در آن سایز دسته های کوچک یا همان mini-batch ها برابر با ۱ است.

در زمینهSGD ، "Minibatch"به این معنی است که گرادیان در کل دسته قبل از به‌روزرسانی وزن‌ها محاسبه می‌شود. اگر از "minibatch"استفاده نمی کنید، هر نمونه آموزشی در یک "batch" پارامترهای الگوریتم یادگیری را به طور مستقل به روز می کند.

معمولا اندازه ی mini-batch برای بیش از ۲۰۰۰ داده بهتره بین ۶۴ تا ۵۱۲ باشه. به طور کلی بهتره از توان های عدد ۲ باشه به خاطر چیدمان و دسترسی سریعتر و بهتر به ممووری و اطمینان از اینکه این دسته های کوچک در CPU یا GPU قرار میگیرند.

برای آماده‌سازی مینی‌بچ‌ها، یکی از مراحل پیش‌پردازش را به‌کار می‌برد: تصادفی کردن مجموعه داده برای تقسیم تصادفی مجموعه داده و سپس تقسیم کردن آن به تعداد تکه‌های مناسب.

درک تفاوت بین این الگوریتم های بهینه سازی ضروری است، زیرا آنها یک تابع کلیدی برای شبکه های عصبی را تشکیل می دهند. به طور خلاصه، اگرچه Batch GD دقت بالاتری نسبت به Stochastic GD دارد، دومی سریعتر است. حد وسط این دو و بهترین مورد، Mini-batch GD، هر دو را برای ارائه دقت خوب و عملکرد خوب ترکیب می کند.

۱-بخش دوم: بهبود SGD با مومنتوم:

مشکل SGD:

نزول گرادیان تصادفی(SGD) پارامترها را برای هر مشاهده به روز می کند که منجر به تعداد بیشتری به روز رسانی می شود. بنابراین این یک رویکرد سریعتر است که به تصمیم گیری سریعتر کمک می کند.

به دلیل رویکرد حریصانه، فقط گرادیان را تقریب (استوکاستیک) می کندیعنی دقیق نیست و تقریبا همگرا نمیشود.

به دلیل نوسانات مکرر، بیش از حد به حداقل هزینه ی مورد نظر ادامه خواهد داد.

برای فهم مومنتوم موردی را در نظر بگیرید که برای رسیدن به مقصد مورد نظر خود به طور مداوم از شما خواسته می شود که همان جهت را دنبال کنید و هنگامی که مطمئن شدید که مسیر درست را دنبال می کنید، قدم های بزرگ تری بردارید و همچنان در همان جهت حرکت می کنید.

اگر گرادیان برای درازمدت در یک سطح صاف باشد، به جای برداشتن گام‌های ثابت، باید گام‌های بزرگ‌تری بردارد و حرکت را ادامه دهد. این رویکرد به عنوان نزول گرادیان مبتنی بر تکانه شناخته می شود.

الگوریتم شیب نزولی با مومنتوم سریعتر از شیب نزولی استاندارد است و دقت آن هم از شیب نزولی تصادفی بیشتر است.

ایده: محاسبه ی میانگین وزنی نمایی گرادیان ها و استفاده از آن برای به روزرسانی وزن ها ی شبکه

این کار کمک میکنه در مراحل ابتدایی که از مقصد دور تر هستیم نوسانات کمتر باشه و سریعتر به نقطه ی موردنظر نزدیک شیم و در نواحی نزدیک سرعت کم بشه و نوسانات بیشتر بشه و با دقت بیشتری به سمت محل مینیمم و نقطه ی مورد نظر حرکت کنیم.

قانون به‌روزرسانی گرادیان نزولی مبتنی بر مومنتوم برای پارامتر وزن:

$$\boldsymbol{w_{t+1} = w_t - v_t}$$
$$\textit{where, } \boldsymbol{v_t = \gamma . v_{t-1} + \eta \nabla w_t}$$

پارامتر گاما γ عبارت حرکتی است که میزان شتاب مورد نظر را نشان می دهد. در اینجا همراه با گرادیان فعلی η∇w(t)، حرکت نیز مطابق تاریخ انجام می‌شود γV(t−1)بنابراین به‌روزرسانی بزرگ‌تر می‌شود که منجر به حرکت سریع‌تر و هم‌گرایی سریع‌تر می‌شود.

v(t) مجموع وزنی به صورت نمایی در حال فروپاشی است، با افزایش t γV(t-1) کوچکتر و کوچکتر می شود، یعنی این معادله به روز رسانی های دورتر را با قدر کوچک و به روز رسانی های اخیر را با قدر زیاد نگه می دارد.

اگر مسأله رو به یک مثال فیزیکی تشبیه کنیم که یک کاسه ی بزرگ داریم که قراره یک توپ از لبه ی آن به نقطه ی وسط در کف آن برسد و بایستد، برای اینکه توپ در نقطه ی وسط کاسه ثابت بماند و دوباره به بالا برنگردد و دقیقا هم در آن همان نقطه بایستد به نیروی اصطکاک نیاز داریم که در لبه ی کاسه کم و هرچه به مرکز نزدیک شویم زیاد شود ضریب γ هم چون کمتر از یک است نقش همین نیروی نگه دارنده را بازی میکند. و مشکل SGD که در آن قدم ها قدر بزرگ بود و تغییرات زیاد اتفاق میافتاد و همگرا نمیشد تا حد خوبی حل میشود.

منبع: لینک های قرار داده شده در پی دی اف سوالات، کورس دوم اسپشیالیزیشن NLP در کورسرا هفته ی اول و دوم

۲- الف) در این سوال قرار است خروجی مختصات پنج نقطه ی تصویر شامل دو چشم، بینی و دو گوشه ی دهان باشد. با توجه به اینکه تصویر دوبعدی است مختصات هر نقطه دو مولفه ی xوy را دارد. اگر در شبکه ی عصبی ای که میخواهیم برای این سوال ارائه دهیم برای هر مولفه در هر نقطه یک نورون در نظر بگیریم، میتوانیم بگوییم لایه ی آخر این شبکه که همان لایه ی خروجی است ۱۰ = ۵ × ۲ نورون نیاز دارد.

با توجه به ویژگی های نقاط موردنظر در تصویر xوy این نقاط به هم وابستگی دارند و برای بهینه کردن تابع ضرر و به طور کلی شناسایی و تشخیص نقاط لازم است که در خروجی دو نورون مربوط به یک نقطه را در یک گروه قرار دهیم و بررسی کنیم.

برای تابع فعال سازی میتوانیم از sigmoid یا tanh استفاده کنیم اما با توجه به دامنه ی تابع sigmoid قبل از استفاده باید مختصات نقاط را normal کنیم. X یا طول نقاط را بر طول تصویر تقسیم کنیم و y یا عرض نقاط را بر عرض تصویر تقسیم کنیم. در هنگام گرفتن خروجی هم حواسمان باشد که مقدار داده شده نرمال شده است و باید آن را به مختصات اصلی با ضرب کردن در طول یا عرض تبدیل کرد و مقایسه کرد یا روی تصویر نشان داد.

تابع فعال سازی ReLU هم میتواند مناسب باشد اما چون مقادیر منفی نداریم مثل تابع خطی عمل میکند و اینکه مشتق در نقاط مختلف آن با هم تفاوتی ندارد و در نتیجه نیازی هم به نرمال کردن نیست.

باتوجه به نکته ای که بالاتر درمورد وابستگی مختصات یک نقطه به طور همزمان هم طول و هم به عرض نقطه بیان کردیم باید تابع ضرری در نظر بگیریم که هم x و هم y نقطه ای که شبکه پیشنهاد میدهد در محاسبه ی آن شرکت داشته باشند.

مثلا یک گزینه میتواند فاصله ی نقطه ی به دست آمده از نقطه ی حقیقی مورد نظر تقسیم یک مقدار مشخص:

$$loss = \frac{\sqrt{(x - x')^2 + (y - y')^2}}{l}$$

این تابع برای زمانی که داده ها نرمال شده باشند مناسب تر است اما برای زمانی که داده ها نرمال نشده اند میتوانیم از تابع زیر استفاده کنیم که د رآن میشود ضرر کل خروجی ها را با هم حساب کنیم:

$$loss = \frac{1}{5}\sum_{i=1}^{5} \frac{\alpha(x_i - x'_i)}{W} + \frac{\beta(y_i - y'_i)}{H}$$

۲-ب)

این مدل با ابزار کراس برای محلی سازی برجسته ی چهره بر اساس اضافه کردن لایه ی رگرسیون نهایی پیاده سازی شده است.دیتاست آن شامل ۳۷۵۵ تصویر چهره است که ۷۶ نقطه از هر تصویر برچسب گذاری شده اند. باری اینکه مشکل حافظه و مموری نداشته باشیم اندازه ی طول و عرض تصویرها تقسیم بر دو شده اند و از ۶۴۰*۴۸۰ به ۳۲۰*۲۴۰ رسیده اند.

مدل:

معماری مدل:

معماری شبکه عصبی در اینجا ResNet50 با وزن از پیش آموزش دیده از ImageNet است. یک جمع بندی میانگین گلوبال برای کاهش بُعد به شبکه اضافه می شود و آخرین لایه رگرسیون با ۱۵۲ خروجی (برای هر یک از ۷۶ نقطه برجسته دو نورون یکی برای x و یکی برای y) و با تابع فعال سازی سیگموئید اضافه می شود. در نتیجه هر دو نورون یک ینقطه ی landmark را با مقداری بین ۰و۱ بدست می آورند چون برد تابع سیگموید بین ۰و۱ است.

در فایل models.py در لینک داده شده جزئیات مدل به صورت آمده است:

```
def get_model(IMAGE_SHAPE):
    with tf.device("/cpu:0"):
        # Define model
        X = Input(shape=IMAGE_SHAPE)
        baseModel = ResNet50(include_top=False)
        pooled = GlobalAveragePooling2D()(baseModel(X))
        dense = Dense(2 * NUM_LANDMARKS, activation="sigmoid", kernel_initializer="glorot_uniform")
        out = dense(pooled)
        model = Model(inputs=X, outputs=out)
    return model
```

تابع ضرر:

تابع از دست دادن مورد استفاده در آموزش میانگین فاصله Euclidean بین نقاط حقیقت زمین و نقاط پیش بینی شده برای هر مینی دسته. نقاطی که توسط انسان نمی توانستند قرار داشته باشد با مختصات (۰، ۰) در مجموعه داده ها نشان داده می شود. این نقاط در محاسبه تابع زیان کنار گذاشته می شوند.

در فایل models.py در لینک داده شده جزئیات تعریف تابع هزینه به صورت زیر آمده است:

```
# Define constant
NUM_LANDMARKS = 76

#Define a loss function where target matrix might have missing values
def mse_with_dontcare(T, Y):
    ##Find the pairs in T having x-y coordinate = (0, 0)
    #Reshape to have x-y coordinate dimension
    T_reshaped = tf.reshape(T, shape=[-1, 2, NUM_LANDMARKS])
    Y_reshaped = tf.reshape(Y, shape=[-1, 2, NUM_LANDMARKS])
    #Calculate Euclidean distance between each pair of points
    distance = tf.norm(T_reshaped - Y_reshaped, ord='euclidean', axis=1)

    #If summing x and y yields zero, then (x, y) == (0, 0)
    T_summed = tf.reduce_sum(T_reshaped, axis=1)
    zero = tf.constant(0.0)
    mask = tf.not_equal(T_summed, zero)
    #Get the interested samples and calculate loss with Mean of Euclidean distance
    masked_distance = tf.boolean_mask(distance, mask)
    return tf.reduce_mean(masked_distance)
```

در این تابع ابتدا آرایه ی شامل خروجی ۱۵۲ نورون را تغییر شکل میدهیم(reshape) و ابعاد آن را به (۲و۷۶) تغییر میدهیم تا به ۷۶ دسته ی دوتایی که همان مختصات نقاط هستند تبدیل شود. سپس فاصله ی اقلیدسی نقاط برچسب دار با نقاط به دست آمده محاسبه شده است. برای حالتی که ممکن است نقطه ی (۰و۰) برگردانده شده باشد و حذف تاثیر آن (چون ممکن است در برخی تصاویر بعضی محل های خواسته شده موجود نباشد) یک mask تعریف شده که روی distance بدست آمده قرار میگیرد. تا حالت نقطه ی صفر را از بین ببرد. در نهایت هم از reduce_mean روی جواب آن استفاده کرده است.

آموزش:

مجموعه داده ها به ۷۰٪ (۲۶۲۸ تصویر) برای آموزش و ۳۰٪ (۱۱۲۷ تصویر) برای آزمایش تقسیم شده است. نقاط حقیقی برای اولین بار توسط عرض و ارتفاع تصویر قبل از تغذیه به شبکه عادی می شوند. برای نتایج در بخش بعدی، پارامترهای آموزشی به صورت زیر است:

Batch size:16

Optimizer:Adam

Learning rate:0.001

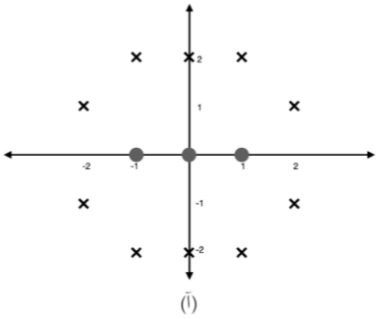
Number of epochs:8

۳- یک پرسپترون تنها میتواند داده هایی که به صورت خطی قابل جدا کردن هستند را دسته بندی کند. توضیح دهید که Madaline چگونه مسئله ی دسته بندی را برای داده هایی که غیر خطی هستند حل میکند.

در مدل Madaline هر یک Adaline به صورت موازی یک خط را با توجه به نقاط ورودی مشخص میکند. در نهایت با اشتراک گیری ناحیه های مشخص شده توسط تقاطع هر خط تولید شده، این مدل میتواند برای تشخیص نواحی محدب یا غیر خطی استفاده بشود. در واقع باید از یک یونیت AND بین خروجی Adaline ها استفاده کرد. شرط جواب دادن این مدل این است که خطوط تقاطع داشته باشند و ناحیه ی محدب شکل بگیرد و قابل تفکیک و جداشدنی باشند.

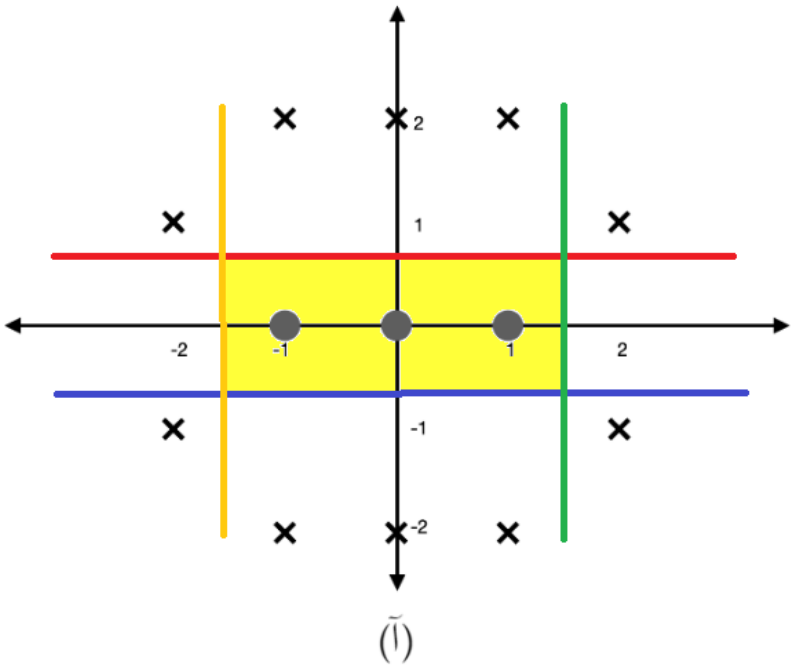
۳- آیا می توان دیاگرام های زیر را توسط Madaline دسته بندی کرد؟ توضیحات لازم را ارائه دهید و در صورت امکان پذیری دسته بندی، معماری شبکه عصبی خود را شرح دهید.

آ)



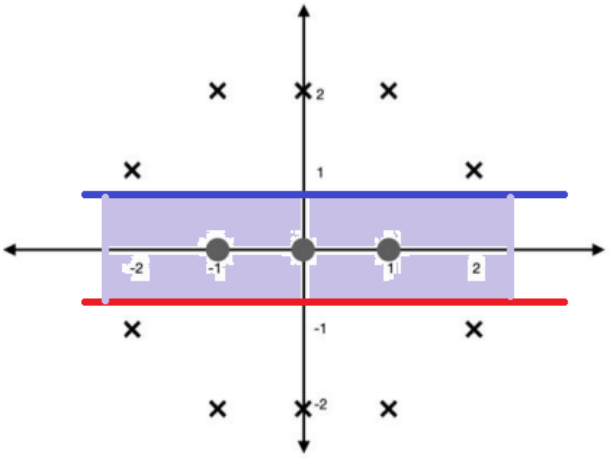
(آ)

در شکل بالا ۴ خط لازم است که نقاط دایره ای را از ضربدر جدا کند ۴ خط یعنی ۴ adaline در آخر هم باید جواب ادلاین ها با هم and شود تا شکل زیر حاصل گردد که در آن ناحیه ی محدب تشکیل شده زرد رنگ شده است.



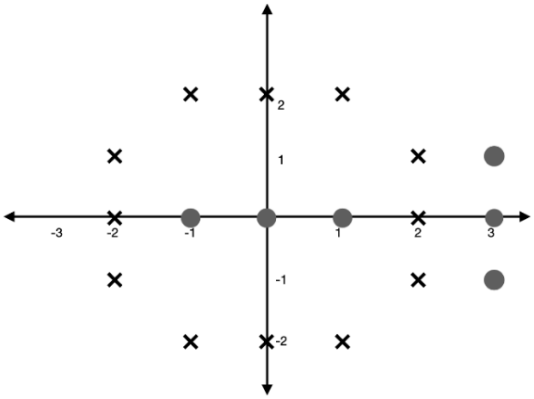
(آ)

میشود یک مدل دیگه هم در نظر گرفت که از دو ادلاین استفاده شود و دو خط تفکیک کننده داشته باشیم:



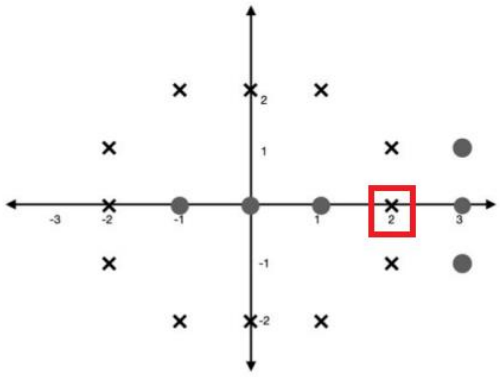
(أ)

ب)



(ب)

در این شکل نقاط تفکیک پذیر با یک ناحیه ی محدب نیستند. ۳ نقطه ی دایره ای در وسط و ۳ نقطه ی دیگر در سمت راست کنار تصویر قرار دارد. اگر مستطیل وسط تصویر را مثل بخش قبل ناحیه ی مدنظر درنظر بگیریم ۳ نقطه بیرون آن هستند و اگر مستطیل دیگری در سمت راست در نظر بگیریم نقاط دایره ای وسط صفحه خارج از آن هستند. در نتیجه در اینجا Madaline کارنمیکند. نقطه ی مشخص شده در شکل زیر ما بین نقاط دایره ای قرار دارد و مانع از تفکیک پذیر بودن مسئله میشود.



(ب)

۴-الف) بنظر من الگوریتم پرسپترون چون ابتدایی ترین روش است و راهکاری برای کاهش ارور ندارد کمترین قابلیت تعمیم را دارد. بعد از آن روش adaline چون با معرفی تابع ارور و روش گرادیان برای کاهش ارور تلاش کرده وضعیت بهتری دارد. بعد از آن روش Madaline که بخاطر موازی اجرا شدن سریعتر است و چون چند الگوریتم ادلاین اجرا میشود میتواند با خطای کتری نقاط مختلف در فضا را کلاس بندی کند. اما روش MLP بهترین روش است و میتواند جداسازی در فضا را به خوبی انجام دهد. هر چه تعداد اشکال بیشتر باشد میتوانیم از تعداد لایه های بیشتری استفاده کنیم و عملا مسئله ی حل نشده ای باقی نمیماند.

در مقایسه ی پرسپتورون و ادلاین تفاوت اصلی در این است که پرسپترون برای به دست آوردن خطا و آپدیت کردن وزن ها از فضای باینری و کلاسیفیکیشن استفاده میکند و ادلاین از فضای پیوسته استفاده میکند. این عملکرد ادلاین باعث میشود در به روز رسانی وزن ها اورهایی که بدست می آید به ارور واقعی نزدیک تر باشد چون مقادیر پیوسته هستند و تقریب نمیخورند. باری همین قابلیت تعمیم ادلاین بیشتر است.

Madaline چون تعداد خطوط بیشتری را تفکیک میکند میتواند برای مسائل پیچیده تر استفاده شود و برای همین قابلیت تعمیم بیشتری دارد و MLP بخاطر استفاده از توابع فعال سازی غیر خطی برای کلاسیفیکیشن های خیلی پیچیده تر هم میتواند کمک کننده باشد و قابلیت تعمیمش از همه بیشتر است.

۴-ب) وقتی یک مدل عملکرد خوبی هنگام اجرا روی train set داشته باشد و عملکرد ضعیفی روی test set داشته باشد میگوییم مدل ما دچار overfitting شده است چون خوب تعمیم داده نشده یا جنرالایز نشده است. این حالت به معنی داشتن واریانس بالا میباشد و تشخیص آن با مقایسه ی خطای مجموعه ی آموزشی و آزمایشی انجام میشود. معمولا در شرایطی این اتفاق می افتد که برای حل سوال ساده از شبکه های خیلی عمیق با تعداد نورون خیلی زیاد استفاده میکنیم. این کار باعث میشود مدل داده ها را حفظ کند.

۴-ج)راه حل ها:

1 -دیتا را زیاد کنید بسته به مسئله و مدل دارد ولی گویا مدل هایی که موجود هستند انقدر پارامتر دارند که باید یک میلیون عکس بدهید

2- drop out -بعد لایه هایconv

3 - regularization l2

4- data augmentation یعنی تصاویری که دارید را عوض کنید. مثلا در دیتاهای تصویری میتوان با چرخاندن چند درجه یه تصویر یا قرینه ی افقی کردن تصویر، تصویر جدیدی برای دیتاست تعریف کرد. یا در داده های متنی از کلمات هم معنی استفاده کرد و متن جدید تولید کرد. هر دفعه که به شبکه داده میدهید و اینگونه شبکه یک چیز را یاد نمیگیره و overfit شدن را برای شبکه سخت میکنید.

۵- پاسخ این سوال در فایل Q5 قرار داده شده است.

برای این سوال از یک مدل پرسپتورون از نوع batch gradient descent استفاده کردیم. این شبکه در لایه ی خروجی خود از تابع sigmoid استفاده میکند چون در تابع Nor دو نوع خروجی ۰ یا ۱ داریم این بهترین انتخاب است.

Q5

calling libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

input and output for Nor function

In [2]:
```
x_train = np.array([
  [0.0, 0.0, 1.0, 1.0], # Input feature x1
  [0.0, 1.0, 0.1, 1.0], # Input feature x2
])

y_train = np.array([
  [1.0, 0.0, 0.0, 0.0]
])
```

این شبکه ی عصبی یک لایه با دو نورون دارد در نتیجه w1و w2 و یک b داریم که باید در ابتدا به صورت رندوم مقداردهی شوند. سپس در هر تکرار با تابع predict خروجی نورون ها با کمک این وزن ها محاسبه میشوند. سپس costو loss را بدست می آوریم و دقت را تا اینجای کار حساب میکنیم سپس وزن ها را بر اساس مقدار ارور و هزینه و نرخ یادگیری آپدیت میکنیم.

```
np.random.seed(1)

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

def predict(x, w, b):
  return sigmoid(np.dot(w, x) + b)

def CEE(y_predict, y):
  m = y.shape[1]
  loss = -1 * (y * np.log(y_predict) + (1 - y) * np.log(1 - y_predict))
  cost = (1 / m) * np.sum(loss)
  return loss, cost

def MSE(y_predict, y):
  m = y.shape[1]
  loss = 1/2 * (y_predict - y)**2
  cost = (1 / m) * np.sum(loss)
  return loss, cost

def computeAccuracy(y_predict, y):
  return 100 - np.mean(np.abs(y_predict - y)) * 100
```

```python
def perceptron(x, y, learning_rate, iterations, iter_log):
  feat, m = x.shape
  # Init parameters
  # In 1 layer NN initializing parameters to 0 or random doesn't matter
  w = np.random.randn(1, feat) * 0.01
  b = np.random.randn() * 0.01
  costs = []
  accuracies = []
  # Batch Gradient Descent
  for i in range(iterations):
    # Forward propagation
    y_predict = predict(x, w, b)
    # Cost and Accuracy
    difference = y_predict - y
    loss, cost = CEE(y_predict, y)
    costs.append(cost)
    accuracy = computeAccuracy(y_predict, y)
    accuracies.append(accuracy)
    # Backward propagation
    dw = difference * x
    db = difference
    dW = (1 / m) * np.sum(dw, axis=1)
    dB = (1 / m) * np.sum(db, axis=1)
    # Update parameters
    w -= learning_rate * dW
    b -= learning_rate * dB
    # Print cost and accuracy
    if (not iter_log == 0) and (i % iter_log == 0 or i == iterations - 1):
      print('Iteration:', i, '  cost:', cost, ' accuracy:', accuracy)
  report = {
      'costs': costs,
      'accuracies': accuracies}
  return w, b, report
```

در نهایت نتیجه ی شبکه به صورت زیر خواهد شد:

using Model

```python
LEARNING_RATE = 0.3
N_EPOCH = 1500
REPORT_MOD = 150

w, b, report = perceptron(x_train, y_train, LEARNING_RATE, N_EPOCH, REPORT_MOD)
y_predict = predict(x_train, w, b)

print("w: ", w)
print("b: ", b)
print("Predicted Output: ", np.around(y_predict, 3))
```

```
Iteration: 0   cost: 0.6870590606375945  accuracy: 50.308206732367445
Iteration: 150   cost: 0.1588000604951101  accuracy: 86.10577139826948
Iteration: 300   cost: 0.09117204614499591  accuracy: 91.5475421329291
Iteration: 450   cost: 0.06299474413256158  accuracy: 94.02044961529906
Iteration: 600   cost: 0.04781847936143997  accuracy: 95.40333998180034
Iteration: 750   cost: 0.03841220474071571  accuracy: 96.27862057966048
Iteration: 900   cost: 0.03204021874157451  accuracy: 96.87952022952184
Iteration: 1050   cost: 0.02745077754157378  accuracy: 97.31633156531402
Iteration: 1200   cost: 0.02399372301237016  accuracy: 97.647593283616
Iteration: 1350   cost: 0.021299169031722007  accuracy: 97.90712279196521
Iteration: 1499   cost: 0.019154664890597372  accuracy: 98.11450962897297
w:  [[-6.49301357 -7.19968137]]
b:  [3.11912118]
Predicted Output:  [[0.958 0.017 0.016 0.   ]]
```

۶-پاسخ این سوال در فایل Q6 قرار داده شده است.

در مرحله ی اول کتابخانه ها و ابزار و دیتاست لازم را import کردیم.

calling libraries

```
from keras import Model
from keras.layers import *
from keras.optimizers import *
from keras.models import Sequential
import keras
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
```

سپس از آنجایی که در MLP ورودی ها باید یک بعد داشته باشند تصاویر را ازشکل ۲۸*۲۸ به ۷۴۸ تبدیل کردیم. سپس با توجه به اینکه y_train در mnist مقادیر عددی هستند آنها را به categorical data تبدیل کردیم.

Data Set

```
class_number = 10
image_number = 784

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

train_dims = x_train.shape[0]
test_dims = x_test.shape[0]

x_train = x_train.reshape((train_dims, image_number))
x_test = x_test.reshape((test_dims, image_number))

y_train = keras.utils.to_categorical(y_train, class_number)
y_test = keras.utils.to_categorical(y_test, class_number)
```

سپس مدل MLP را به صورت زیر تعریف کردیم که ۳ لایه ی میانی داشته باشد که در آنها تابع فعال سازیReLU و در لایه ی خروجی softmax است. لایه ی visible که لایه ی اول است ۷۸۴ پیکسل دریافت میکند پس ۷۸۴ یونیت دارد. برای لایه ی میانی اول ۳۹۲و لایه ی میانی دوم ۱۹۶و لایه میانی سوم ۹۸ یونیت درنظر گرفتم و درنهایت هم چون ۱۰ کلاس مختلف خواهیم داشت لایه ی خروجی ۱۰ یونیت دارد.

تابع هزینه را هم cross-entropy درنظر گرفتیم.

سپس مدل را روی دیتا ی آموزشی فیت کردیم.

تعداد epoch ها را ۲۵ درنظر گرفتم.

creating a Multiple Layer Perceptron

```
visible = Input(shape = x_train[0].shape)
layer = Dense(units = 392, activation = 'relu')(visible)
layer = Dense(units = 196, activation = 'relu')(layer)
layer = Dense(units = 98, activation = 'relu')(layer)
layer = Dense(units = class_number, activation = 'softmax')(layer)
model = Model(inputs = visible, outputs = layer)

model.summary()

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

history = model.fit(x_train, y_train,
                    batch_size=250,
                    epochs=25,
                    validation_data=(x_test, y_test))

print(history.history.keys())
```

سپس آن را روی دیتای آزمایشی امتحان نمودیم.

```
score = model.evaluate(x_test, y_test, verbose=1)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.1919 - accuracy: 0.9722
Test score: 0.19194093346595764
Test accuracy: 0.9721999764442444
```

همانطور که مشخص است هم مقدار loss و هم مقدار دقت عالی است.

سپس نمودار های دقت و خطا را برای هر iteration رسم کردیم: