به نام خدا

نام: فاطمه زهرا بخشنده **استاد: دکتر ناصر مزینی**

شماره دانشجویی: 98522157 **درس مبانی هوش محاسباتی**

گزارش تمرین 1:

سوال اول:

روش Gradient Descent الگوریتمی است که در شبکه خود استفاده می کنیم و به کمک آن وزن ها را در راستای بهینه شدن تابع ضرر به طورر مکرر آپدیت می کنیم، و به این صورت در جهت min شدن تابع Loss حرکت می کنیم.

ترتیب مراحل به این صورت است که: ابتدا Forward Propagation انجام داده و خروجی را پیش بینی می کنیم. سپس یک تابع ضرر با توجه به اختلاف خروجی پیش بینی شده و خروجی واقعی حساب می کنیم.

سپس Backward Propagation انجام می دهیم که gradients را برمی گرداند.

حالا با استفاده از gradient descent وزن ها و بایاس را به صورت زیر آپدیت می کنیم:

$$W := W - \alpha \frac{\partial J}{\partial W}$$

$$b := b - \alpha \frac{\partial J}{\partial b}$$

Batch Gradient Descent: در این روش، هر epoch با استفاده از تمام مجموعه داده‌های آموزشی اجرا می‌شود و پس از آن ضرر را محاسبه کرده و مقادیر W و b را به‌روزرسانی می کند. اگرچه این روش همگرایی پایدار و یک خطای پایدار را فراهم می کند، اما چون از کل مجموعه آموزشی استفاده می کند، برای مجموعه داده های بزرگ بسیار کند است.

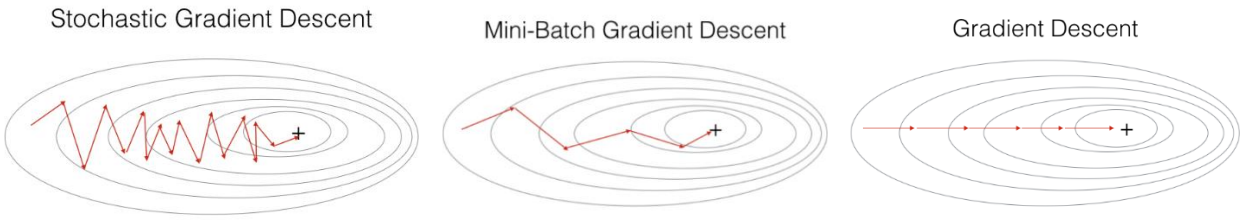
Mini Batch Gradient Descent: در این روش مجموعه داده خود را به چند دسته (batch) تقسیم می‌کنیم. بنابراین به جای اینکه منتظر بمانیم تا الگوریتم در هر epoch برای کل مجموعه داده اجرا شود، ابتدا mini batch ها به ترتیب وارد شبکه می شوند، و هر بار با استفاده از یکی از این mini batch ها پارامتر ها را را بروزرسانی می کنیم. این روش باعث می شود تا به سرعت به سمت global minimum در تابع ضرر حرکت کنیم و وزن‌ها را چندین بار در هر epoch به‌روزرسانی کنیم. رایج ترین اندازه ها برای

mini batch ها 16، 32، 64، 128، 256 و 512 هستند. بیشتر پروژه ها نیز از Mini-batch GD استفاده می کنند زیرا در مجموعه داده های بزرگتر سریع تر است.

Stochastic gradient descent: در این روش هر mini batch برابر با یک نمونه از مجموعه آموزشی است. مثلا اولین mini batch برابر با اولین مثال آموزشی است.

$$(x^{\{1\}}, y^{\{1\}}) = (x^{(1)}, y^{(1)})$$

نویزی بودن این روش با تنظیم learning rate قابل حل است اما نقطه ضعف این روش در این است که مزیت استفاده از vectorization را از دست می دهیم، نوسان بیشتری نیز دارد اما سریعتر همگرا می شود.



مقایسه:

GD:

مزایا: مقدار Loss یکنواتر است. احتمال عبور از نقطه global minimum کمتر است. برای داده هایی با حجم کم مناسب است. از vectorization نهایت استفاده را می برد.

معایب: زمان بیشتری برای محاسبه گرادیان نیاز دارد. به تعداد epoch های بیشتری نیاز دارد. برای داده هایی با حجم بالا مناسب نیست. حافظه بیشتری نیاز دارد. احتمال گیر کردن در نقطه local minima بیشتر است. دیرتر converge می شود.

SGD:

مزایا: مقدار گرادیان سریعتر محاسبه می شود. حافظه کمتری اشغال می کند. احتمال گیر کردن در نقطه local minima کمتر است. سریعتر converge می شود.

معایب: مزیت استفاده از vectorization را از دست می دهیم. رویکرد حریصانه دارد و گرادیان را تقریب می زند. مقدار Loss نوسان زیادی دارد. ممکن است از نقطه global minimum عبور کند.

Momentum یکی دیگر از الگوریتم های optimization است که از سرعت نیز بهره می برد. در واقع در هنگام آپدیت وزن ها به گرادیان های گذشته نیز نگاه می کند. اگر گرادیان برای درازمدت در یک جهت باشد، به جای برداشتن گام های ثابت، گام‌های بزرگ‌تری بر می دارد و حرکت را ادامه دهد. این الگوریتم حرکت را در جهت گرادیان های ثابت اضافه می کند و اگر شیب ها در جهت های مختلف باشند، Momentum را لغو می کند. در واقع پارامتر ها را به صورت زیر آپدیت می کند:

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

گاما یک هایپرپارامتر است که معمولا از مقدار 0.9 برای آن استفاده می شوند.

با استفاده از Momentum، learning را در یک راستا آرامتر و در راستایی که گرادیان با v هم جهت است تند تر می کنیم. در نتیجه مقدار نوسان های ما کم می شود و سرعت الگوریتم برای رسیدن به نقطه minimum را بهبود می بخشیم. و با توجه به در نظر گرفتن شتاب، دیگر در saddle points متوقف نمی شودیم و از آن عبور می کنیم. از local optima ها هم راحت تر می تواند عبور کند و احتمال گیر کردن آن کمتر است. و به دلیل در نظر گرفتن اصطکاک در global optimum نیز راحتتر converge می شود. پس توانسته تا حد خوبی دو مشکل SGD را حل کند. (نوسان زیاد و عبور از global minimum)

منابع: لینک و لینک

سوال دوم:

الف) می توان با این مسئله به عنوان یک مسئله localization نگاه کرد که با مسائل کلاس بندی یا رگرشن متفاوت است.

1- تعداد نورون های لایه آخر: به ازای هر نقطه، دو نورون (یکی برای x و یکی برای y) در نظر میگیریم که در مجموع 10 نورون ($2 \times Num\ of\ Landmarks$) در لایه آخر خواهیم داشت. در واقع انگار 2 دسته 5 تایی داریم، چون هر دو نورون به هم وابسته اند و مختصات یکی از نقاط را نشان می دهند.

2- تابع فعالسازی: یک رویکرد این است که به صورت Linear Regression از تابع فعالسازی استفاده نکنیم و تنها Logits را در نظر بگیریم. در این صورت از تابع ضرر MSE استفاده می کنیم. اما در این حالت مختصات‌های x و y را بدون توجه به همدیگر بهینه می‌کنیم و انگار هیچ دسته بندی ای انجام نداده ایم، پس شاید خیلی رویکرد مناسبی نباشد.

رویکرد دوم این است که داده ها را نرمال کرده و x و y آن ها را به Height و Width تقسیم کنیم. در این حالت مختصات نقاط به بازه (0, 1) می روند. سپس در لایه خروجی از تابع Sigmoid استفاده می کنیم تا مقادیر خروجی هم به بازه (0, 1) بروند و با مقادیر واقعی در بازه یکسانی باشند. یا از تابع فعالسازی Tanh استفاده می کنیم و مقادیر خروجی هم به بازه (-1, 1) Scale می شوند. و یا در این حالت هم می‌توانیم از تابع فعالسازی استفاده نکنیم، آنگاه نورون‌های لایه آخر هر مقداری می‌توانند داشته باشند.

3- تابع ضرر: طبق تابع فعالسازی که انتخاب می کنیم با توجه به توضیحات مورد قبل اگر رویکرد اول را انجام دهیم می توانیم از MSE استفاده کنیم. اما با توجه به این که در این مسئله با مختصات نقاط سر و کار داریم، و همچنین با توجه به مقاله ذکر شده در منبع، بهتر است از تابع ضرری مبتنی بر فاصله نقاط استفاده کنیم. می‌توانیم فاصله $L1$ یا $L2$ را در نظر بگیریم. همچنین با توجه به تابع فعالسازی و Scale کردن یا نکردن می‌تواند ثابت‌های متفاوتی را نیز داشته باشد. برای مثال:

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

که در آن (x, y) مختصات واقعی نقطه است، (x′, y′) مختصات تشخیص خروجی مدل است و l اندازه border است. در واقع با در نظر گرفتن تابع ضرری به این شکل که مبتنی بر فاصله است، آن دسته بندی که در مورد های قبل در مورد آن صحبت کردم، حفظ می شوند و هر دو نورون که در یک دسته قرار دارند و مربوط به یک نقطه در تصویر هستند، به صورت وابسته به هم آپدیت خواهند شد تا فاصله نقطه تشخیص داده شده کم کم با نقطه واقعی کاهش یابد.

پس این می تواند بهترین رویکرد برای این مسئله باشد.

ب) در این کد NUM_LANDMARKS = 76 است و به جای 5 نقطه در واقع 76 نقطه Landmark داریم. از شبکه ResNet50 به عنوان Base Model استفاده شده است و دیتاست مورد نظر ImageNet است.

1- تعداد نورون های لایه آخر: 76 نقطه داریم پس لایه خروجی طبق توضیحات قسمت الف 152 نورون دارد.

2- تابع فعالسازی: از sigmoid استفاده شده است و مختصات x و y را به بازه Scale (0, 1) کرده است. سپس در پایان برای اینکه مختصات نقاط به Scale واقعی رفته و روی تصویر اصلی نمایش داده شود آن هارا در Width و Height تصویر ورودی ضرب کرده است. (در فایل test.py)

3- تابع ضرر: طبق توضیحات قسمت الف، با توجه به این که در این مسئله با مختصات نقاط سر و کار داریم، از تابع ضرری مبتنی بر فاصله بین نقاط استفاده کرده است.

در این کد نام تابع محاسبه گر ضرر را mse_with_dontcare قرار داده است. که در آن از فاصله اقلیدسی $L2$ برای Loss استفاده شده است. برای انجام این کار ابتدا نورون ها و اعداد واقعی را Reshape می کند تا به 76 دسته دوتایی با مقادیر بین 0 و 1 تبدیل شود. سپس فاصله اقلیدسی را بین نقاط پیشبینی شده Y_reshaped و نقاط واقعی T_reshaped حساب می کند.

```
#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)
```

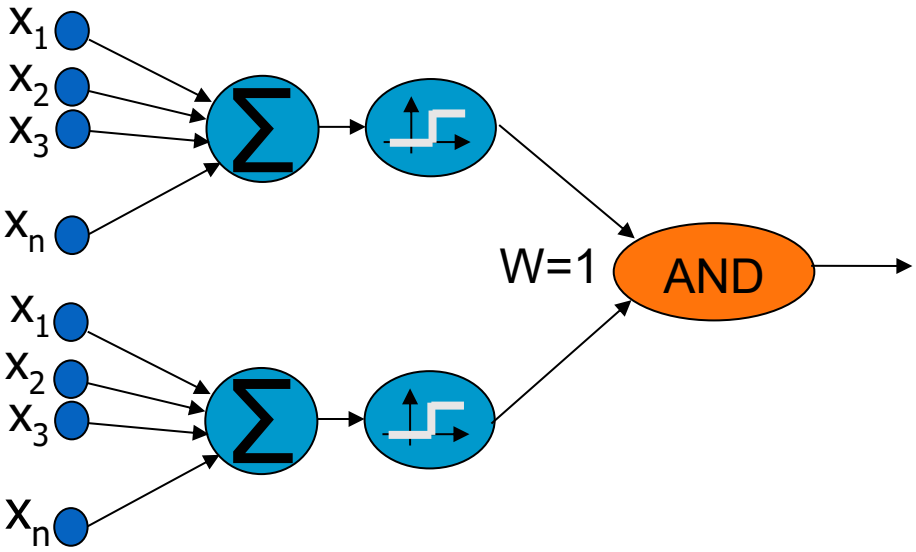
اگر بعضی از Landmark ها مشخص نشده باشند در محاسبه Loss آن ها را در نظر نمی گیرد و با ایجاد یک Mask از مقادیر واقعی اثر این نقاط را از بین می برد. سپس مثل mse میانگین Loss را حساب می کند و برمی گرداند.

```
#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)
```

منابع: لینک و لینک

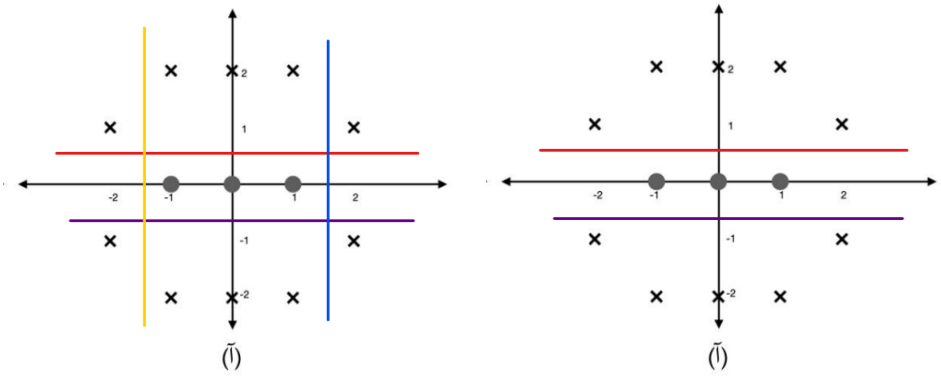
سوال سوم:

در MAdaline در واقع از چند Adaline باهم به صورت موازی استفاده می کنیم که هرکدام یک خط را تشخیص می دهند. (Adaline همان Perceptron می باشد که تنها روش آپدیت کردن وزن ها در آن کمی متفاوت است و از Delta Rule استفاده می کند). در نهایت خروجی همه این Adaline ها باهم اشتراک گیری و and می شوند، و اینگونه یک ناحیه غیرخطی را یاد می گیرد.

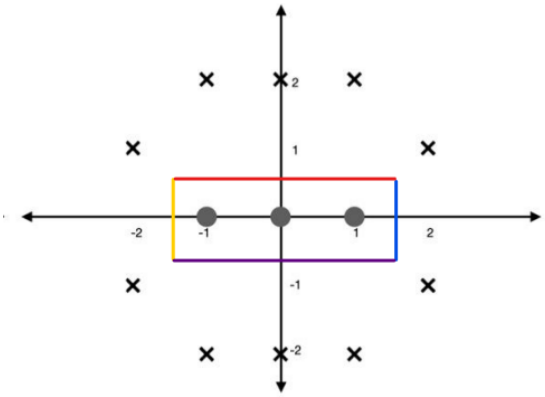


شرط اینکه مسئله توسط MAdaline قابل حل باشد این است که ناحیه ها جداپذیر باشند و یک ناحیه Convex داشته باشیم. یعنی ناحیه ای که میخواهیم جدا کنیم یک منحنی باشد که زاویه داخلی بزرگتر از 180 درجه نداشته باشد.

الف) می توان با MAdaline این مسئله را حل کرد و ضربدرهارا از نقاط تفکیک کرد. (هرچقدر دقیقتر شویم و منحنی به دایره ها نزدیکتر شود overfit بیشتر میشود و دقیقتر داده های آموزشی را تفکیک می کند). مثلا با 2 یا 4 Adaline به صورت زیر نقاط تفکیک می شوند. در واقع هر Adaline یک خط پیدا می کند و بین این خط ها اشتراک گرفته می شود.



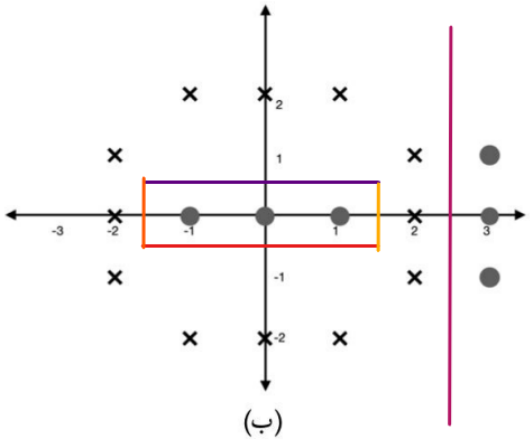
منحنی اشتراک خط ها وقتی با 4 تا Adaline آن را حل کنیم یک مستطیل به شکل زیر خواهد شد:



(آ)

ب) این مسئله با MAdaline قابل حل **نیست** زیرا هیچ منحنی Convex ای برای یافتن یک ناحیه وجود ندارد و هر حالتی که در نظر بگیریم زاویه بیشتر از 180 درجه بوجود می آید.

در واقع این مسئله را می توان با MLP به صورت زیر حل کرد:



(ب)

در این مسئله در واقع اشتراک چند تا Adaline کافی نیست بلکه باید از لایه مخفی نیز استفاده کنیم. در واقع یک لایه حداقل با 5 نورون برای پیدا کردن Convex های مختلف، و یک لایه اضافه برای and کردن نواحی Convex می خواهیم.

منابع: اسلاید

سوال چهارم:

1- بنظرم این شبکه ها از نظر قابلیت تعمیم به ترتیب به صورت زیر هستند:

MLP > MAdaline > Adaline > Perceptron

دلیل: Perceptron یک شبکه ساده است که وزن هارا با استفاده از خروجی باینری و گسسته آپدیت می کند. یعنی با استفاده از آخرین خروجی پس از مقایسه با treshhold.

Adaline یک پله از Perceptron پیشرفته تر است چون وزن هارا با استفاده از خروجی پیوسته، یعنی خروجی تابع f قبل از مقایسه با treshhold، آپدیت می کند که باعث می شود error به صورت دقیقتری محاسبه شده و وزن ها بهتر در راستای نزدیکتر شدن خروجی پیش بینی شده به خروجی اصلی آپدیت شوند. پس قابلیت تعمیم Adaline بیشتر از Perceptron است.

MAdaline چند لایه است و از چند Adaline به صورت موازی استفاده می کند و بین آن ها اشتراک می گیرد، در واقع Adaline قابلیت تفکیک خطی دارد اما MAdaline قابلیت تفکیک به صورت یک منحنی Convex را دارد، پس قابلیت تعمیم آن نسبت به Adaline و Perceptron بیشتر است.

در MLP تعداد نورون ها و تعداد لایه ها نامحدود است در واقع می توانیم از لایه های مخفی و توابع فعالسازی غیرخطی استفاده کرده و ناحیه بندی های پیچیده تری شامل Convex های مختلف داشته باشیم و بسیاری از مسائل را با آن حل کنیم. پس قابلیت تعمیم آن از همه موارد پیشین بیشتر است.

2- Overfit زمانی اتفاق می افتد که یک مدل دیتای آموزشی را به خوبی یاد گرفته و در واقع حفظ می کند. اما دقت آن روی داده validation و test خیلی خوب نیست و با دقت آن روی داده train فاصله زیادی دارد (واریانس بالایی داریم). در واقع مدل جزئیات داده آموزشی را حفظ می کند و سعی کرده روندی را در داده ها پیش بینی کند که بیش از حد نویزی است و مختص داده آموزشی است. اما این قضیه روی عملکرد مدل تاثیر منفی دارد چون واقعیت موجود در همه داده هارا منعکس نمی کند. overfit شدن یک شبکه می تواند یک سری دلیل داشته باشد. مثلا سایز دیتاست کم است یا دیتا ها نویزی هستند. یا تعداد لایه ها و نورون ها زیاد هستند و شبکه برای مسئله ما بیش از حد پیچیده است.

3- اگر شبکه ما دچار overfit شود و واریانس بالایی داشته باشیم می توانیم از روش های زیر استفاده کنیم:

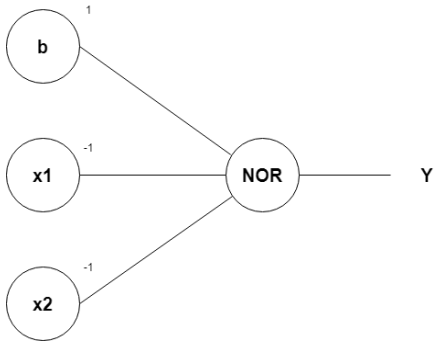
- استفاده از دیتای بیتشر
- Regularization، که خود، روش های متفاوتی را شامل می شود:
  - استفاده از روش L1 یا L2 که منجر به اضافه کردن هزینه به Loss function برای آپدیت وزن ها می شود و وزن ها کوچکتر می شوند. (Weight decay)

- استفاده از Drop out که در هر iteration نود های مختلفی صفر می شوند و در واقع در هر iteration شبکه ما به یک صورت متفاوت کوچکتر می شود. (در موقع test، Drop out نداریم).
- Data augmentation یعنی ایجاد تغییراتی در دیتا مثلا چرخاندن یا زاویه دادن تصاویر، و اضافه کردن این دیتاها به دیتاست.
- Early stopping: یک جایی زودتر تا w ها خیلی بزرگتر نشده اند و شبکه overfit نشده است learning را متوقف کنیم. اما مشکل این کار این است که دو هدف optimize کردن cost function و overfit نشدن را باهم mixکرده ایم و دیگر به صورت مستقل آن ها را انجام نمی دهیم.

منابع: لینک

سوال پنجم:

باید یک perceptron به شکل زیر طراحی کنیم که خروجی آن تنها زمانی 1 می شود که x1 و x2 هردو 0 باشند.



برای این سوال دو مدل Implement انجام دادم.

1- پیاده سازی Perceptron ساده:

در تابع perceptron_1 یک پرسپترون ساده دقیقا مانند اسلاید هارا پیاده سازی کردم.

- Initialization: set **w**(0) =0
- Activation: activate perceptron by applying input example (vector **x**(n) and desired response d(n))
- Compute actual response of perceptron:

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$

- Adapt weight vector: if d(n) and y(n) are different then

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta[d(n)-y(n)]\mathbf{x}(n)$$

Learning rate ← η ; $d(n)-y(n)$ = $\delta(w,x)$

$$\text{Where } d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \in C_1 \\ -1 & \text{if } \mathbf{x}(n) \in C_2 \end{cases}$$

- Continuation: increment time step n by 1 and go to Activation step

24

این پرسپترون دقیقا طبق الگوریتم این اسلاید و طبق مثال های کلاس عمل می کند، ابتدا delta را برابر اختلاف خروجی پیش بینی شده و خروجی اصلی گذاشته و وزن ها را با استفاده از delta و etha که همان learning rate است، آپدیت می کنیم. در این پیاده سازی الگوریتم را تا جایی ادامه می دهیم که بیشترین تغییرات از یک عدد خاص کمتر شود. هرچه این عدد را کوچک تر بگیریم زمان الگوریتم طولانی تر و خروجی دقیقتر خواهد شد. خروجی وزن ها برای این پیاده سازی به صورت زیر شد:

```
w: [ 0.49621838 -0.99546834 -0.99797661]
```

یعنی تقریبا b = 0.5 و w1 = -1 و w2 = -1 شد که اگر آن را برای همه مثال ها چک کنیم می بینیم درست است.

2- پیاده سازی پیشرفته تر:

برای این مدل پیاده سازی توابع متعددی مثل sigmoid و propagate و optimize و predict تعریف کردم و از آن ها در تابع perceptron_2 استفاده کردم. در این پیاده سازی از تابع فعالسازی sigmoid و تابع ضرر cross entropy استفاده کردم که همگی تنها با استفاده از numpy نوشته شده است. همچنین پیاده سازی به صورت vectorized انجام شده است.

مراحل پیاده سازی به صورت زیر است:

Forward Propagation:

$$Z = W^T X + b$$

$$A = \sigma(Z)$$

$$J = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}))$$

Backward Propagation:

$$\frac{\partial J}{\partial W} = \frac{1}{m} X (A - Y)^T$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)})$$

Updating parameters:

$$W := W - \alpha \frac{\partial J}{\partial W}$$

$$b := b - \alpha \frac{\partial J}{\partial b}$$

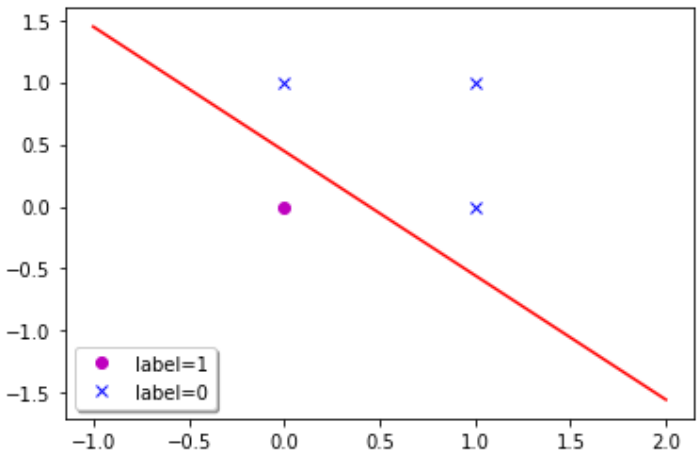
گزارش آموزش مدل ساخته شده با این پیاده سازی به صورت زیر است:

```
iteration 0: Cost: 0.693147 , Accuracy: 50.000000
iteration 100: Cost: 0.156254 , Accuracy: 86.277764
iteration 200: Cost: 0.089756 , Accuracy: 91.663089
iteration 300: Cost: 0.062008 , Accuracy: 94.106566
iteration 400: Cost: 0.047058 , Accuracy: 95.472059
iteration 500: Cost: 0.037793 , Accuracy: 96.335765
iteration 600: Cost: 0.031518 , Accuracy: 96.928386
iteration 700: Cost: 0.026999 , Accuracy: 97.358979
iteration 800: Cost: 0.023596 , Accuracy: 97.685402
iteration 900: Cost: 0.020945 , Accuracy: 97.941060
iteration 1000: Cost: 0.018822 , Accuracy: 98.146537
iteration 1100: Cost: 0.017085 , Accuracy: 98.315180
iteration 1200: Cost: 0.015638 , Accuracy: 98.456011
iteration 1300: Cost: 0.014415 , Accuracy: 98.575344
iteration 1400: Cost: 0.013368 , Accuracy: 98.677722
iteration 1500: Cost: 0.012461 , Accuracy: 98.766499
iteration 1600: Cost: 0.011669 , Accuracy: 98.844202
iteration 1700: Cost: 0.010970 , Accuracy: 98.912771
iteration 1800: Cost: 0.010350 , Accuracy: 98.973720
iteration 1900: Cost: 0.009795 , Accuracy: 99.028245
train accuracy: 99.028245169548
```

همچنین وزن ها و خروجی پیش بینی شده به صورت زیر هستند:

```
w: [[-8.66182298] [-8.66182298]]
b: 3.865753849376571
Predicted Output: [[1 0 0 0]]
```

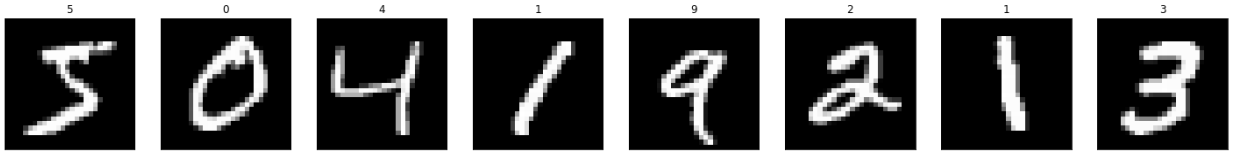
3- در نهایت برای اطمینان از درستی آموزش پرسپترون، نموداری شامل خط متمایز کننده ای که توسط پرسپترون آموخته شده است می‌کشیم. نقاط دایره نقاط با label = 1 و نقاط ضربدر نقاط با label = 0 هستند که این برچسب ها در legend نمودار هم مشخص شده اند.



منابع: لینک و لینک

سوال ششم:

دیتاست MNIST شامل اعداد دست نویس 0 تا 9 می باشد پس با یک مسئله classification ده کلاسه رو به رو هستنیم و لایه آخر باید 10 نورون داشته باشد. بخشی از دیتاست را نشان می دهیم:



X_train را با تقسیم بر 255، normalize می کنیم. Y_train را با استفاده از تابع to categorical و روش one hot کد می کنیم، تا قابل استفاده در شبکه multi class باشد.

با استفاده از keras یک مدل sequential میسازیم. این مدل یک لایه Input دارد که روی آن لایه Flatten میگذاریم. دو لایه مخفی با 128 نورون و تابع فعالسازی ReLU و یک لایه خروجی با 10 نورون به تعداد کلاس ها با تابع فعالسازی Softmax استفاده می کنیم. ساختار مدل به صورت زیر است:

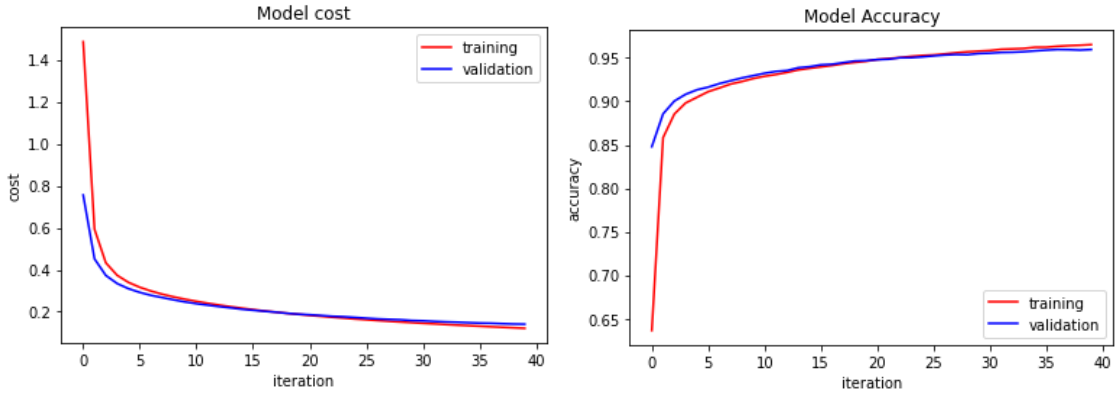
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 activation (Activation)     (None, 128)               0

 dense_1 (Dense)             (None, 128)               16512

 activation_1 (Activation)   (None, 128)               0

 dense_2 (Dense)             (None, 10)                1290

 activation_2 (Activation)   (None, 10)                0

=================================================================
Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0
```

از mini_batch با سایز 128 استفاده می کنیم تا هر epoch خیلی طول نکشد. مدل را با 40 epoch فیت می کنیم. چون مسئله classification است از تابع خطای Categorical Cross Entropy استفاده می کنیم. برای جلوگیری از overfit شدن مدل نیز مقدار Validation Split را 0.2 می گذاریم تا از بخشی از داده Train را برای Validation استفاده کنیم.

در آخر مدل را evaluate می کنیم که خروجی آن برای داده train و test به صورت زیر است:

```
Train loss: 0.12364466488361359 Train accuracy: 0.9647833108901978
Test loss: 0.13685601949691772 Test accuracy: 0.9599000215530396
```

نمودار دقت و خطا به ازای هر iteration:



منابع: لینک