دانشگاه صنعتی شریف

دانشکده مهندسی مکانیک

شبکه‌های عصبی مصنوعی

گزارش تمرین ۱

رباتیک اجتماعی و شناختی

مریم کریمی جعفری، ۹۹۱۰۶۶۱۷

استاد

دکتر طاهری

بهار ۱۴۰۳

فهرست

سوالات تشریحی

۱)

الف) روش بهینه‌سازی AdaDelta

همانطور که می‌دانید برای آپدیت پارامترها از روابط زیر استفاده می‌شود:

$$x_{t+1} = x_t + \Delta x_t$$

$$\Delta x_t = -\eta g_t$$

که این درواقع همان رابطهٔ زیر است:

$$\Longrightarrow W \leftarrow W - \eta \nabla C$$

که x همان پارامترها، g_t همان گرادیان‌ها و η همان نرخ آموزش است که باید طوری مشخص شود که نه خیلی زیاد باشد که گرادیان‌ها واگرا شوند و نه خیلی کوچک باشد که سرعت آموزش را کم کند.

روش بهینه سازی AdaDelta، برای هر بعد پارامترها، یعنی وزن‌ها و بایاس‌ها، یک نرخ آموزش جدا در نظر می‌گیرد. علاوه بر این، در این روش نیازی به تعیین نرخ آموزش به صورت دستی نیست و میزان effective learning rate در طول فرایند آموزش، محاسبه و دچار تغییر می‌شود. این روش برگرفته از روش AdaGrad است که با ایجاد تغییراتی در آن، عیوب آن را تا حدودی برطرف کردند. همچنین در تولید این روش از روش‌هایی نظیر Momentum (فرمول این روش به $\Delta x_t = \rho \Delta x_{t-1} - \eta g_t$ صورت است) و روش‌هایی مثل روش نیوتن که از اطلاعات مربوط به مشتق‌های مرتبه دوم استفاده می‌کند (در این روش از هسیان مشتق‌های مرتبه دوم و یا مقادیر تقریبی آن‌ها استفاده می‌شود)، اقتباس شده است اما فقط از مشتق‌های مرتبه اول استفاده می‌شود.

در ابتدا لازم است مختصری بر روش AdaGrad داشته باشیم. رابطهٔ روش AdaGrad به صورت زیر است:

$$\Delta x_t = -\frac{\eta}{\sqrt{\Sigma_{\tau=1}^{t} g_\tau^2}} g_\tau$$

در این رابطه η ، global learning rate است که باید انتخاب شود. با این حال با توجه به مخرج برای هر پارامتری یک نرخ آموزش (dynamic rate) جدا محاسبه می‌شود که با گرادیان‌ها رابطهٔ عکس دارد. در مخرج $norm2$ برای همه گرادیان‌های قبلی محاسبه می‌شود. با این رابطه نرخ آموزش در گذر زمان کاهش می‌یابد، اما مشکلاتی دارد که AdaDelta برای برطرف‌کردن آن‌ها به وجود آمده است. یکی از آن‌ها این است که مخرج ممکن است صفر شود، یا اینکه به سمت بی‌نهایت میل کند. علاوه بر این، در AdaGrad بر خلاف روش‌هایی

مثل نیوتن که از مشتق دوم استفاده می‌کنند، واحد (unit) تغییرات پارامترها با خود پارامترها یکسان نیست. علاوه بر آن ما همچنان به انتخاب η نیاز داریم.

پیشنهاد AdaDelta برای جلوگیری از انباشت گرادیان در مخرج و به سمت بی‌نهایت میل کردن آن:

می‌توان به جای در نظر گرفتن همهٔ گرادیان های قبلی، بخشی از آن ها را به سایز w در نظر گرفت، اما چون این روش ها کارآیی کافی را نداشته است، از یک نوع میانگین گرادیان‌ها که به صورت نزولی کاهش می‌یابد (exponentially decaying average of the past gradients)، استفاده می‌شود. از آنجایی که نیاز به توان دوم داریم تا مشابه فرمول قبل در آید، میانگین تبدیل به RMS می‌شود. این میانگین برای زمان t به صورت زیر است:

$$\mathrm{RMS}[g]_t = \sqrt{E[g^2]_t + \epsilon}$$

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$$

کاربرد ρ مشابه کاربرد آن در روش Momentum است. ϵ مقداری است که برای جلوگیری از صفرشدن مخرج استفاده شده است. به ازای $t = 0$، مقدار $E[g^2]_t$ را برابر صفر مقداردهی اولیه می‌کنیم. رابطه بعد از این اصلاح، به صورت زیر در می‌آید:

$$\Delta x_t = -\frac{\eta}{\mathrm{RMS}[g]_t} g_\tau$$

پیشنهاد AdaDelta برای جایگزینی η و اصلاح واحد:

همانطور که گفته شد، باید واحد را نیز اصلاح می‌کردند. پس، از روش نیوتن و روش‌های مشابه که از هسیان مشتق دوم‌ها استفاده می‌کردند و واحد درستی داشتند، کمک گرفتند.

واحد در روش‌هایی مثل SGD، Momentum و AdaGrad:

$$\text{units of } \Delta x \propto \text{units of } g \propto \frac{\partial f}{\partial x} \propto \frac{1}{\text{units of } x}$$

واحد در روش‌هایی مثل Newton:

$$\Delta x \propto H^{-1} g \propto \frac{\frac{\partial f}{\partial x}}{\frac{\partial^2 f}{\partial x^2}} \propto \text{units of } x$$

رابطهٔ روش Newton به صورت زیر است:

$$\Delta x_t = H_t^{-1} g_t$$

همچنین داریم:

$$\Delta x = \frac{\frac{\partial f}{\partial x}}{\frac{\partial^2 f}{\partial x^2}} \Longrightarrow \frac{1}{\frac{\partial^2 f}{\partial x^2}} = \frac{\Delta x}{\frac{\partial f}{\partial x}}$$

یعنی برای به کارگیری مشتق دوم و روش Momentum، کافی است از ترکیب آپدیت پارامترها در صورت و گرادیان در مخرج استفاده کرد که از آنجایی که از قبل مخرج ما با گرادیان‌ها مرتبط بود، کافی است در صورت به جای η ، ترمی مرتبط به مقدار آپدیت پارامترها یا همان Δx_t گذاشت. پس از RMS استفاده می‌شود. از آنجایی که Δx_t در همین لحظه را نمی‌دانیم، از یک لحظه قبل استفاده می‌کنیم.

رابطه در نهایت به صورت زیر می‌شود:

$$\Delta x_t = -\frac{RMS[\Delta x]_{t-1}}{RMS[g]_t} g_\tau$$

اگرچه ما همچنان نیاز به تعیین مقادیر ϵ و ρ داریم. الگوریتم این روش به صورت زیر است:

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ
Require: Initial parameter x_1
1: Initialize accumulation variables $E[g^2]_0 = 0$, $E[\Delta x^2]_0 = 0$
2: **for** $t = 1 : T$ **do** %% Loop over # of updates
3: Compute Gradient: g_t
4: Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$
5: Compute Update: $\Delta x_t = -\frac{RMS[\Delta x]_{t-1}}{RMS[g]_t} g_t$
6: Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2$
7: Apply Update: $x_{t+1} = x_t + \Delta x_t$
8: **end for**

مقایسه‌ای که روی داده‌های mnist انجام شده است:

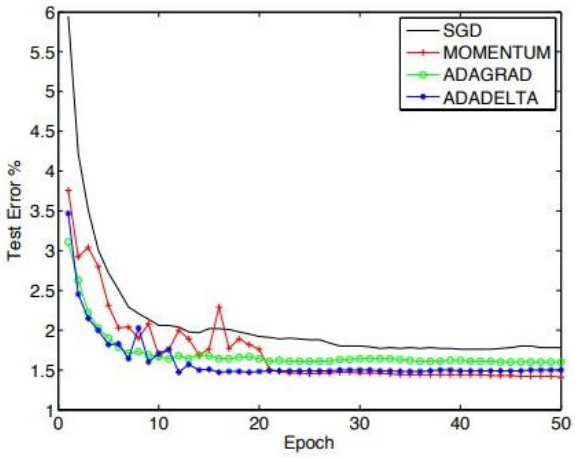


Fig. 1. Comparison of learning rate methods on MNIST digit classification for 50 epochs.

منابع:

- Zeiler, Matthew D. "Adadelta: an adaptive learning rate method." arXiv preprint arXiv:1212.5701 (2012).
- Milman, Oren (2018). Understanding the mathematics of AdaGrad and AdaDelta. https://datascience.stackexchange.com/questions/27676/understanding-the-mathematics-of-adagrad-and-adadelta.

- برای فهمیدن نکات مبهم مقاله از ChatGPT استفاده کردم.

ب) L2 Regularization

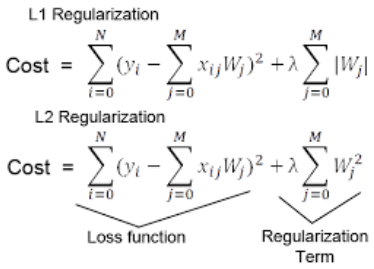
L2 Regularization از جمله روش‌هایی است که برای افزایش generalization استفاده می‌شود و به جلوگیری از overfitting کمک می‌کند و پیچیدگی مدل را کاهش می‌دهد. به طور کلی این روش جزو روش‌هایی است که مقداری را تحت عنوان penalty، به مقدار loss اضافه می‌کنند که برای مثال، تابع وزن‌های شبکه است. برای بهتر فهمیدن مفهموم L2 Regularization به مقایسه آن به L1 Regularization می‌پردازیم. همچنین قابل ذکر است که روش‌هایی ترکیبی از این دو نیز وجود دارد.

L1 Regularization (Lasso)

این روش مجموعه‌ای از قدر مطلق پارامترها را به loss اضافه می‌کند. یعنی تا حدودی تاثیر وزن‌هایی را که مقدار خیلی کمی دارند و ویژگی‌های متناظر آن‌ها را، صفر می‌کنند که این خود نوعی feature selection است. این روش پیچیدگی مدل را با در نظر نگرفتن featureهایی که تاثیر زیادی ندارند، کم می‌کند. همچنین مدل را به مدلی با وزن‌های غیر صفر کمتری، تبدیل می‌کند.

L2 Regularization (Ridge)

این روش مجموع مربع پارامترها را به loss اضافه می‌کند. این روش برخلاف L1، مقدار وزن‌ها را صفر در نظر نمی‌گیرد بلکه مقدار و تاثیر آن‌ها را خیلی کم می‌کند. با این روش تاثیر یک ویژگی ممکن است به تاثیر چند ویژگی تقسیم شود و feature selection صورت نمی‌گیرد. زیرا که ممکن است رابطه‌ای بین featureها باشد که با روش L1 این در نظر گرفته نمی‌شود. یعنی در واقع این روش پیچیدگی مدل را با کاهش همه ضرایب، کم می‌کند.



λ پارامتری است که نیاز به مشخص کردن و نوعی hyper parameter tuning دارد. این پارامتر مقدار تاثیر وزن‌ها در loss را مشخص می‌کند. (regularization parameter)

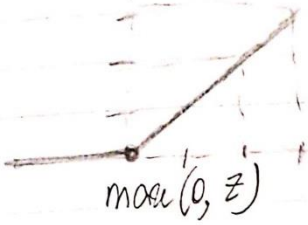
منابع:

- Google (2022). Regularization for Simplicity: L2 Regularization. https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/l2-regularization
- Otten, Neri Van (2023). L1 And L2 Regularization Explained, When To Use Them & Practical How To Examples. https://spotintelligence.com/2023/05/26/l1-l2-regularization/#:~:text=L2%20regularization%20distributes%20the%20impact,of%20models%20by%20reducing%20overfitting.

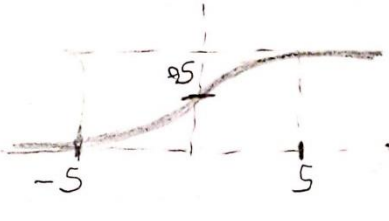
پ)

سوال های تشریحی

ReLU

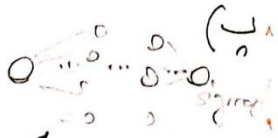


Sigmoid

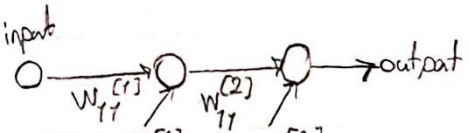


$\frac{1}{1+e^{-z}}$

ب)



این یک مسئله رگرسیون است.



$w_{11}^{[1]}$ $w_{11}^{[2]}$ $b_1^{[1]}$ $b_1^{[2]}$

$w_{11}^{[1]} = 0.5$ $w_{11}^{[2]} = 0.3$

$b_1^{[1]} = 0.07$ $b_1^{[2]} = 1$

input	actual_output
0,2	0,550
0,4	0,599
0,6	0,646
0,8	0,690
1	0,731

$z_1^{[1]} = w_{11}^{[1]} x + b_1^{[1]} \Rightarrow a_1^{[1]} = \max(0, z_1^{[1]})$

$z_1^{[2]} = w_{11}^{[2]} a_1^{[1]} + b_1^{[2]} \Rightarrow a_1^{[2]} = \hat{y} = \max(0, z_1^{[2]})$

برای داده اول داریم:

$z_1^{[1]} = 0,5 \times 0,2 + 0,07 = 0,17 \Rightarrow a_1^{[1]} = 0,17$

$z_1^{[2]} = 0,17 \times 0,3 + 1 = 1,033 \Rightarrow \hat{y} = a_1^{[2]} = 1,033$

$loss = (\hat{y} - y_{actual})^2 = 0,2508$

برای سایر sample ها نیز این محاسبات را انجام دهید، خواهید داشت:

$x = 0,4 \Rightarrow \hat{y} = 1,063 \Rightarrow loss = 0,2387$

$0,6 \Rightarrow 1,093 \Rightarrow 0,224$

$0,8 \Rightarrow 1,123 \Rightarrow 0,1864$

$1 \Rightarrow 1,153 \Rightarrow 0,2245$

backpropagation

$$\frac{\partial loss}{\partial w_{11}^{[2]}} = \begin{cases} 2(\hat{y} - y_{actual}) \times a_1^{[1]} & (\text{if } z_1^{[2]} > 0) \\ 0 \end{cases}$$

$$\frac{\partial loss}{\partial w_{11}^{[1]}} = \begin{cases} \frac{\partial loss}{\partial a_1^{[1]}} (*) \times x & (\text{if } z_1^{[1]} > 0) \\ 0 \end{cases}$$

$$(*)\ \frac{\partial loss}{\partial a_1^{[1]}} = \begin{cases} 2(\hat{y} - y_{actual}) \times w_{11}^{[2]} & \text{if } (z_1^{[2]} > 0) \\ 0 \end{cases}$$

$$\frac{\partial loss}{\partial b_1^{[2]}} = \begin{cases} 2(\hat{y} - y_{actual}) & (\text{if } z_1^{[2]} > 0) \\ 0 \end{cases}$$

$$\frac{\partial loss}{\partial b_1^{[1]}} = \begin{cases} \frac{\partial loss}{\partial a_1^{[1]}} & (\text{if } z_1^{[1]} > 0) \\ 0 \end{cases}$$

برای همه سمپل ها خواهیم داشت :

x	$\frac{\partial loss}{\partial w_{11}^{[2]}}$	$\frac{\partial loss}{\partial w_{11}^{[1]}}$	$\frac{\partial loss}{\partial b_1^{[2]}}$	$\frac{\partial loss}{\partial b_1^{[1]}}$
0,2	0,1063	0,05796	0,966	0,2898
0,4	0,19488	0,11136	0,928	0,2784
0,6	0,27714	0,16092	0,894	0,2682
0,8	0,35506	0,20784	0,866	0,2598
1	0,43044	0,2532	0,844	0,2532
sum	1,36382	0,79132	4,498	1,3308

وزن‌ها را با این مقادیر جدید به صورت زیر محاسبه:

$$w_{11}^{[1]} = 0,5 - 0,1 \times 0,79732 = 0,420868$$

$$b_1^{[1]} = 0,01 - 0,1 \times 1,3308 = -0,12308$$

$$w_{11}^{[2]} = 0,3 - 0,1 \times 1,36382 = 0,163618$$

$$b_1^{[2]} = 1 - 0,1 \times 4,498 = 0,5502$$

حال به محاسبهٔ خروجی شبکه، loss، با این وزن‌های جدید می‌پردازیم:

برای $x = 0,2$ داریم:

$$z_1^{[1]} = 0,2 \times 0,420868 - 0,12308 = 0,065672$$

$$a_1^{[1]} = 0,065672$$

$$z_1^{[2]} = 0,2 \times 0,163618 + 0,5502 = 0,560776$$

$$\hat{y} = a_1^{[2]} = 0,560776$$

$$\Rightarrow \underbrace{loss}_{MSE} = (\hat{y} - y_{actual})^2 = 0,00071278$$

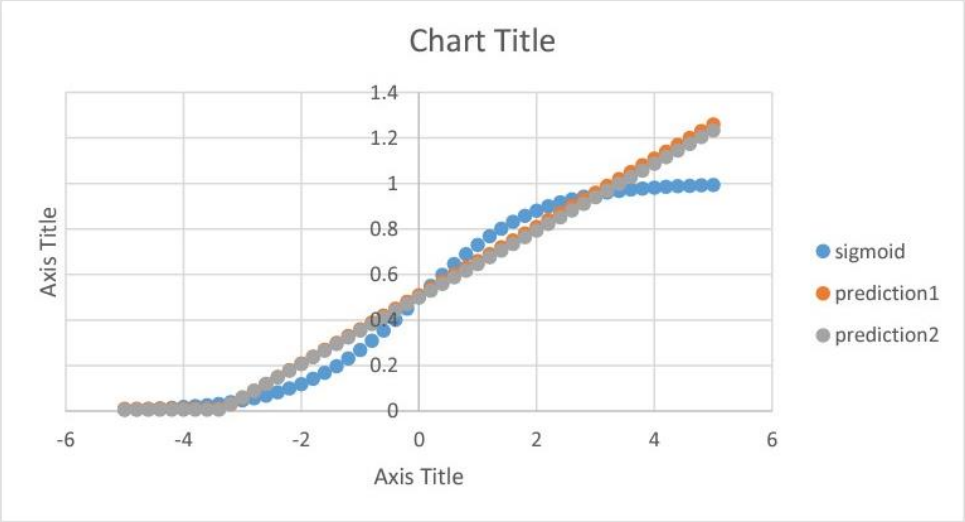
برای باقی سمپل‌ها خواهیم داشت:

x	$\hat{y}$	loss
0,4	0,560837	0,0014918
0,6	0,560946	0,0072911
0,8	0,561061	0,016399
1	0,567175	0,028982

$$Loss = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_{actual_i})^2$$

$$\Rightarrow Loss = 0,01085$$

مشتق در مبدا چیزی حدود ۱ است و کران ما در $+\infty$ به بی‌نهایت میل می‌کند. به نظر می‌رسد مدل انتخاب شده برای x های بیشتر از ۵ تخمین درستی ندارد.



ت) hyper parameters tuning

هایپر پارامترها، پارامترهایی هستند که توسط مدل یافت نمی‌شوند و باید آن‌ها را در ابتدا مقداردهی کرد؛ مثل نرخ آموزش و یا اینکه چه تعداد نرون و چه تعداد لایه برای شبکه عصبی‌مان می‌خواهیم بگذاریم. فرایند tune کردن این هایپرپارامترها یعنی بهترین مقادیری را که موجب کمترین loss و دقت بالایی می‌شوند، بیابیم. روش‌هایی برای tune کردن وجود دارد که در ادامه به توضیح تعدادی از آن‌ها می‌پردازیم:

Manual search

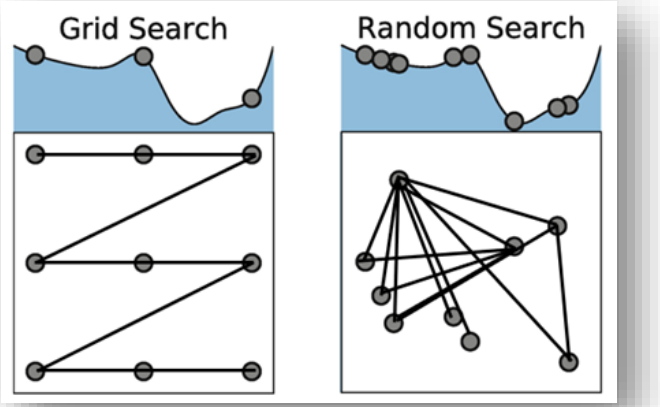
در این روش برنامه نویس به صورت دستی مقادیر هایپرپارامترها را تعیین می‌کند. مثل کدزدن ما که مقدار نرخ آموزش را مشخص می‌کنیم و با تغییر آن، مقداری را که موجب دقت ماکسیمم می‌شود را بیابیم. پس این روش برای تعداد کم هایپرپارامتر استفاده می‌شود. همانطور که مشخص است، فرایندی زمانبر می‌باشد.

Grid search

این روشی است که از تمامی ترکیب‌های موجود برای هایپرپارامترها استفاده می‌کند، مدل را ساخته و سپس ارزیابی می‌کند. در نهایت با توجه به معیار انتخاب شده مثل دقت، بهترین را انتخاب می‌کند. منظور از ترکیب یعنی ما یک بازه برای هر یک از پارامترها داشته باشیم و برای هر عضوی از آن بازه که با یک گام مشخص جلو می‌رود، هایپرپارامتر انتخاب شود. این فرایندی طولانی است و لود محاسباتی بالایی دارد. علاوه بر آن ما همچنان نیازمند مشخص کردن بازه‌ای هستیم که ممکن است لزوما بهینه‌ترین حالت ممکن در آن بازه‌ها نباشد. اما همچنان برای مدل‌های کوچک قابل استفاده است؛ زیرا کارایی لازم را دارد.

Random search

این روش به طور رندوم و تصادفی، ترکیب گفته شده در روش قبل را انتخاب می‌کند که باعث افزایش سرعت این روش نسبت به روش قبل می‌شود. این روش ممکن است برای مدل‌های بزرگ و پیچیده خوب عمل نکند اما همچنان به دلیل سادگی اعمال می‌شود.



Bayesian search

این روش از روش بهینه‌سازی Bayesian استفاده می‌کند که به طور کلی به دنبال ساخت یک **مدل احتمالاتی** از تابع هدف است. در این روش به دنبال **مدلی** هستیم که عملکرد مدل اصلی را بر حسب هایپرپارامترها تعریف کند. این **مدل** ساخته شده ترکیب هایپرپارامترهای بعدی را برای ما انتخاب می‌کند تا تست کنیم، طوری که به سمت بهترشدن شبکه اصلی می‌رویم. بنابراین **مدل** گفته‌شده نیاز به دانستن هایپرپارامترهای قبلی و نتایج خروجی حاصل از آن‌ها دارد. این روش برای مدل‌های بزرگ هم به کار می‌رود اما به منابع محاسباتی بیشتری نیاز دارد و در کل نسبت به روش‌های قبلی، روشی پیچیده‌تر است.

Halving grid search and random search

در این روش در ابتدا با منابع کم (برای مثال با دیتا سمپل‌های کم) ترکیب‌های مختلف را انتخاب و امتحان و تعدادی از آن‌ها را حذف می‌کنیم. سپس منابع را به طور تدریجی بیشتر کرده و به دنبال بهترین می‌گردیم.

منابع:

- Run.ai (no date). What Is Hyperparameter Tuning. https://www.run.ai/guides/hyperparameter-tuning
- Abhigyan (2021). Different types of Hyper-Parameter Tuning https://medium.com/analytics-vidhya/different-types-of-hyper-parameter-tuning-3d99ca624baa

- Pandian, Shanthababu (2022). A Comprehensive Guide on Hyperparameter Tuning and its Techniques. https://www.analyticsvidhya.com/blog/2022/02/a-comprehensive-guide-on-hyperparameter-tuning-and-its-techniques/

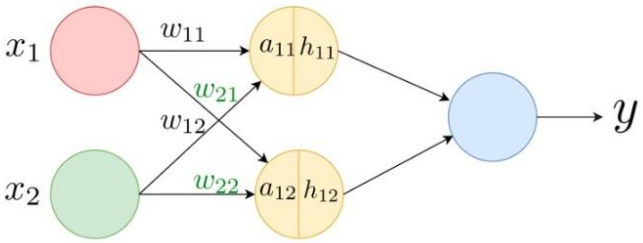
ث) Initialization

1) Zero or Constant Initialization

۱) وزن‌دهی اولیه به همه نرون‌ها به صورت یکسان (صفر یا یک مقدار ثابت)

یعنی همانطور که از اسم روش پیدا است، به همه وزن‌ها و بایاس‌ها یک مقدار داده شود. این کار باعث می‌شود که وزن‌ها به یک شکل آپدیت شده و شبکه را محدود کنند.

اگر شبکه‌ای را به شکل زیر در نظر بگیریم که بایاس‌های آن صفر است:



$$a_{11} = w_{11}x_1 + w_{12}x_2$$
$$a_{12} = w_{21}x_1 + w_{22}x_2$$

برای آپدیت وزن‌ها خواهیم داشت:

$$\nabla w_{11} = \frac{\partial \mathcal{L}}{\partial y} . \frac{\partial y}{\partial h_{11}} . \frac{\partial h_{11}}{\partial a_{11}} . x_1$$
$$\nabla w_{21} = \frac{\partial \mathcal{L}}{\partial y} . \frac{\partial y}{\partial h_{12}} . \frac{\partial h_{12}}{\partial a_{12}} . x_1$$
$$But\ h_{11} = h_{12}\ (since\ a_{11} = a_{12})$$
$$\implies \nabla w_{11} = \nabla w_{21}$$

$$\nabla w_{11} = \frac{\partial \mathcal{L}}{\partial y} . \frac{\partial y}{\partial h_{11}} . \frac{\partial h_{11}}{\partial a_{11}} . x_1$$
$$\nabla w_{22} = \frac{\partial \mathcal{L}}{\partial y} . \frac{\partial y}{\partial h_{12}} . \frac{\partial h_{12}}{\partial a_{12}} . x_2$$
$$But\ h_{11} = h_{12}\ (since\ a_{11} = a_{12})$$
$$\implies \nabla w_{12} = \nabla w_{22}$$

چون وزن‌ها صفر هستند، $a_{11}=a_{12}$ و $h_{11}=h_{12}$ خواهند شد. پس گرادیان‌ها به طور متقارن با هم برابر و آپدیت‌های نظیرشان یکسان خواهند شد. شاید بتوان با روش‌هایی مثل dropout از پدیده symmetry یا همان متقارن‌شدن جلوگیری کرد اما روش‌های دیگری نیز برای مقداردهی اولیه به وجود آمدند.

2) Random Initialization

۲) مقداردهی رندوم با توابع گوسی و یا توزیع یکنواخت

روش دیگری که می‌توان برای مقداردهی اولیه وزن‌ها استفاده کرد که برای بهبود روش قبل به میدان آمد، استفاده از Uniform Distribution و یا Normal Distribution برای وزن‌دهی رندوم، است. یعنی وزن‌ها از یکی از این توزیع‌ها پیروی کنند. برای مثال اگر توزیع گوسی استاندارد باشد، میانگین وزن‌ها صفر است. اما کوچک یا بزرگ بودن این مقدارهای اولیه ممکن است باعث vanishing یا exploding گرادیان شود. یا در تابع‌های فعالسازی مثل tanh، باعث اشباع شدن آن شود.

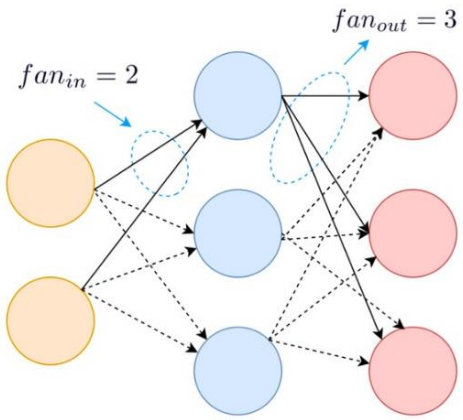
3) Xavier/Glorot Initialization

۳) روش زیویر

در این روش همچنان از توزیع‌های یکنواخت و گوسی استفاده می‌شود اما بازهٔ مناسب آن‌ها پیدا شده است.

Fan_{in} : تعداد ورودی‌های یک نرون

Fan_{out} : تعداد خروجی‌های یک نرون



اگر از توزیع نرمال استفاده شود، وزن‌ها از بازهٔ زیر انتخاب می‌شوند:

$$w_i \sim U[-\sqrt{\frac{\sigma}{fan\_in+fan\_out}}, \sqrt{\frac{\sigma}{fan\_in+fan\_out}}]$$

اگر از توزیع نرمال استفاده شود، داریم:

$$w_i \sim N(0, \sigma)$$

$$\sigma = \sqrt{\frac{6}{fan\_in + fan\_out}}$$

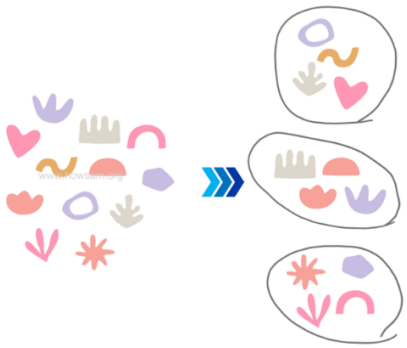
با استفاده از keras می‌توان روش‌های بالا را پیاده سازی کرد و حالت دیفالت آن، همین زیوزر است.

منابع:

- **Priya C, Bala (2023). Weight Initialization Techniques in Neural Networks. https://www.pinecone.io/learn/weight-initialization/**
- **geeksforgeeks (2022). Weight Initialization Techniques for Deep Neural Networks. https://www.geeksforgeeks.org/weight-initialization-techniques-for-deep-neural-networks/**

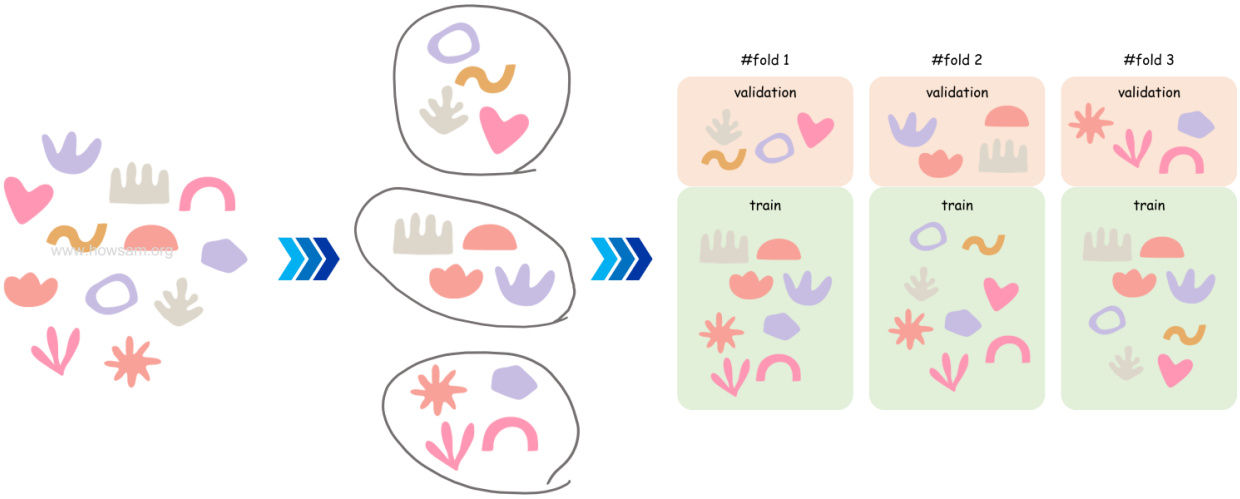
ج) k-fold cross validation

اعتبارسنجی متقابل cross validation: validation به طور کلی برای افزایش تعمیم‌پذیری (generalization) یک مدل استفاده می‌شود. اینکه مدل بتواند برای داده‌های unseen هم به خوبی عمل کند و overfit نشده باشد. اینکه ما در train کردنمان ۲۰ درصد از داده‌ها را برای validation و مابقی را برای train استفاده می‌کنیم نیز خود یک روش CV(cross validation) است. اما روش k-fold به این صورت است که دیتا ست را به k دسته تقسیم می‌کنیم و به هر کدام یک fold می‌گوییم. به تعداد k دور نیز یک ساختار مشخص از مدل را train و validate می‌کنیم. در هر مرحله آموزش، یکی از foldها را برای validation و مابقی را برای train استفاده می‌کنیم. پس هر fold، k بار به عنوان validation و k-1 بار برای train استفاده می‌شود. از هر مرحله آموزش، یک دقت به دست می‌آوریم و میانگین آن‌ها را گزارش می‌دهیم. برای مثال اگر دیتاستی با ۱۲ داده داشته باشیم و آن را به سه قسمت تقسیم کنیم، داریم:



در اینجا k=3 است و ما باید مدل را ۳ بار train کنیم. ۳ دقت از هر train به دست می‌آوریم که میانگین آن‌ها را محاسبه می‌کنیم. به طور معمول مقادیر ۵ و ۱۰برای k بسیار استفاده می‌شوند.

برای دیتاست ۱۲تایی بالا، foldها در نهایت به شکل زیر هستند:



منابع:

- **پیلوا، الهام (۲۰۲۳). روش اعتبارسنجی متقابل یا cross validation چیست. https://howsam.org/validation-techniques-ml/**

- **Lyashenko, Vladimir. Jha ,Abhishek (2023). Cross-Validation in Machine Learning: How to Do It Right. https://neptune.ai/blog/cross-validation-in-machine-learning-how-to-do-it-right**