6. Adaptive learning

Vooruitblik

In dit hoofdstuk bekijken we een aantal varianten op stochastic gradient descent die erop gericht zijn om de convergentie naar optimale modelparameters te bevorderen. We combineren hiertoe ideeën gerelateerd aan (mini)batch learning, learning rate decay, momentum, en schaling van gradiënten om uiteindelijk te komen tot de methode van adaptive moments estimation.

6.1. Minibatch learning

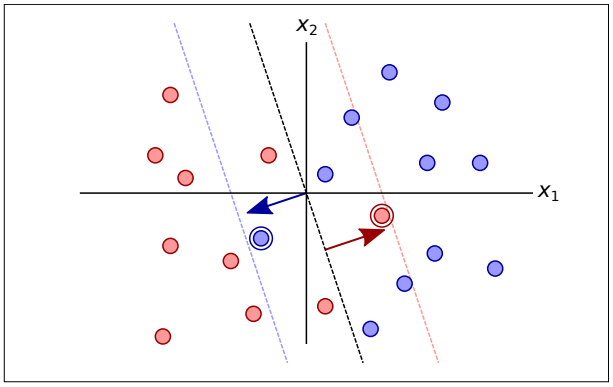
Tot dusverre trainen we het neurale netwerk elke keer met precies één instance tegelijkertijd. Het effect hiervan is dat je het model telkens aanleert om die ene instance goed te kunnen voorspellen. Door de instances snel en willekeurig af te wisselen hoop je dat het model bij een oplossing terecht komt die het voor alle instances redelijk goed doet.

Deze benadering werkt in de praktijk behoorlijk goed. Ook kan worden aangetoond dat voor een voldoende kleine learning rate α dit gegarandeerd leidt tot een optimum. Als je α heel klein kiest werk je de modelparameters immers maar heel weinig bij. Het effect daarvan is dat het model goed in staat is om de geleerde gewichten op grond van de instances die het eerder gezien heeft te "onthouden". Op die manier onthoudt het model effectief alle instances, en optimaliseert het dus ook de prestaties voor alle instances.

Het nadeel van een lage α is natuurlijk echter dat het model slechts heel langzaam leert. Als de learning rate groter wordt gekozen kan het model weliswaar sneller convergeren naar een lage waarde voor de loss l, maar deze is dan voornamelijk gebaseerd op de laatste instances waarmee het model getraind is. Eerdere instances worden dan snel weer "vergeten" omdat de gewichten al heel gauw flink veranderen. Een goede prestatie voor de ene instance hoeft niet ook een goede prestatie voor een andere instance te betekenen. Omdat bij hoge α vooral de meest recente trainingsinstances de uitkomst bepalen en het model dus vooral die instances correct leert voorspellen, maakt dit het soms moeilijk om een model te trainen dat het goed doet voor álle data.

Bekijk bijvoorbeeld eens de figuur hieronder. Hier wordt een situatie getoond waarin op enig moment een model zoals logistische regressie is getraind om de zwart gestippelde scheidingslijn middenin te trekken. Er worden dan twee instances verkeerd geclassificeerd, een blauwe en een rode, beide omcirkeld. Zodra het model op de blauwe misgeclassificeerde instance wordt getraind zal de scheidingslijn naar links worden getrokken. Het hangt ervan af hoe groot de learning rate is hoe ver de lijn verschuift, maar een model dat deze ene instance goed classificeert zal het op de dataset als geheel waarschijnlijk slecht

doen omdat een boel andere instances dan aan de verkeerde kant van de lijn komen te liggen.



Zodra even later het model op de rode omcirkelde instance wordt getraind zal de scheidingslijn terug naar rechts worden getrokken. Opnieuw leidt dat tot een verbetering van het model voor die ene instance, maar kunnen andere instances daardoor verslechteren. Als je telkens maar op één instance tegelijk traint wordt de scheidingslijn numeriek heen en weer bewogen. De oplossing heeft dan sterk de neiging om heen en weer te springen: op zeker moment presteert je model misschien goed op de ene instance maar niet de andere, en even later presteert het dan weer goed op die andere instance maar niet meer op die ene. Je komt dan weliswaar ergens in de buurt van een redelijk model, maar op welk moment je ook stopt zul je niet noodzakelijkerwijs een optimaal model vinden voor alle instances samen.

Dit kan worden opgelost door niet om en om te trainen op één individuele instance, maar op de hele dataset tegelijkertijd. Dit wordt ook wel *batch learning* genoemd omdat je je baseert op een hele trainingsbatch met instances. Elke iteratie van het trainingsproces komt dan in één keer overeen met een epoch. Wanneer je traint op één instance per keer noem je dat *online* learning; dit is soms noodzakelijk als de trainingsinstances één voor één gegenereerd worden en je elke keer onmiddellijk je model bij wil werken.

Tot nu toe probeerden we de loss $l = \sum_m \mathcal{L}(\hat{y}_m; y_m)$ van een enkele instance te minimaliseren. De sommatie m liep hier over de output neuronen van het model. Beter lijkt het nu om te kijken naar de gemiddelde loss van alle instances gezamenlijk. Deze wordt ook wel de *cost* genoemd. Overigens worden de termen loss en cost vaak door elkaar gebruikt. Wij hanteren hier de definitie dat de loss beschrijft hoe uit $\hat{y}_m$ en y_m kan worden berekend hoe slecht de voorspelling van één instance is; de cost beschrijft wat het criterium is dat wordt geminimaliseerd, meestal de gemiddelde of totale loss over vele instances in de trainingsdata (soms met inbegrip van extra termen, zoals voor L_1- of L_2-regularisatie uit het vorige hoofdstuk). Tot dusverre waren die twee begrippen identiek, maar vanaf nu lopen ze uiteen.

We kunnen hier stellen dat de cost berekend wordt als

$$J = \frac{1}{N} \sum_{n=1}^{N} l_n$$

waarbij we middelen over de trainingsinstances genummerd $n = 1$ tot en met N. De stochastic gradient descent regel voor een gewicht w_{ij} wordt dan lichtjes aangepast tot

$$w_{ij} \leftarrow w_{ij} - \alpha \frac{\partial J}{\partial w_{ij}}$$

of een vergelijkbare uitdrukking voor de bias b_j. De gradiënt van de cost is gelijk aan de gradiënten van de losses, gemiddeld over alle instances.

Overigens wordt de cost ook vaak gelijk gesteld aan de totale loss over alle instances, dat wil zeggen de som in plaats van het gemiddelde. Dit scheelt slechts de vermenigvuldigingsfactor $\frac{1}{N}$. Dit leidt niet tot een andere oplossing: het minimum voor de een is ook een minimum voor de ander, hoewel de minima niet even "diep" zijn. Wel heeft het effect op de optimale grootte van de learning rate die gekozen dient te worden.

Wanneer je bij elke update van je model alle instances moet doorrekenen leidt dit tot erg veel rekenwerk. Voor grote datasets kan dit nogal uit de klauwen lopen. Vandaar dat in de praktijk een tussenoplossing wordt gekozen die alle voordelen combineert. De dataset wordt dan opgedeeld in kleinere *minibatches* die wel zoveel instances bevatten dat ze voldoende representatief zijn voor de hele dataset, maar toch ook weer niet zoveel dat er vrijwel identieke instances als het ware "dubbelop" in voorkomen. Hoe groot deze minibatches moeten zijn hangt ervan af hoeveel trainingsdata er beschikbaar zijn en hoe ingewikkeld het model en de verdeling van de data zijn, maar als vuistregel worden vaak enkele tientallen tot honderden instances genomen.

Wanneer kortweg gesproken wordt over batch learning wordt soms ook wel eens minibatch learning bedoeld. Het woordgebruik hieromtrent is niet altijd consequent. Om dit onderscheid expliciet te maken kun je spreken van *full batch learning* als je alle instances tezamen gebruikt. Het woord minibatch slaat wel altijd op slechts een deel van de instances. Kies je de grootte van de minibatch gelijk aan 1 dan verkrijg je online learning.

Bij minibatch learning of online learning maak je gebruik van stochastic gradient descent omdat er een toevalsfactor bij komt kijken die bepaalt op welke deel van de instances per keer getraind wordt: de instances in de minibatch worden willekeurig getrokken uit de trainingsdata, dus de cost die wordt geoptimaliseerd is ook in elke minibatch een klein beetje anders. Bij full batch learning kun je spreken van traditionele gradient descent omdat dan de te optimaliseren cost in alle iteraties identiek blijft.

Minibatch learning heeft ook een paar andere praktische voordelen. Enerzijds wordt het model telkens getraind met net een iets andere verzameling van instances. Vergeleken met (full) batch learning heeft dit als voordeel dat de methode wat ongevoeliger wordt voor overfitting. Bij overfitting wordt het model te sterk aangepast aan de toevallige waarden van de gebruikte trainingsdataset; door de trainingsdata te verdelen in steeds weer andere deelverzamelingen wordt overfitting verminderd. Dit lijkt een beetje op het toevoegen van ruis aan de trainingsdata bij data augmentatie. Anderzijds geeft minibatch learning de mogelijkheid om gebruik te maken van parallellisatie van de berekening. Vergeleken met online learning wordt hetzelfde model immers parallel en identiek doorgerekend voor diverse instances, en dit leent zich goed voor speciale hardware, zoals

een GPU. Dit kan het doorrekenen en optimaliseren van neurale netwerken behoorlijk versnellen.

Opgave 110. *
Leg uit dat je online en batch learning allebei kunt zien als speciale gevallen van minibatch learning.

Opgave 111. *
Waarom kunnen opeenvolgende instances bij online learning niet parallel worden doorgerekend?

Opgave 112. **
Stel dat een optimale learning rate $\alpha = 0.1$ zou gelden als je de cost definieert als de *gemiddelde* loss over 100 instances in een minibatch. Hoe zou je in dezelfde situatie de learning rate moeten kiezen als je de cost had gedefinieerd als de *totale* gesommeerde loss over een minibatch van 100 instances?

Opgave 113. ***
Schets een voorbeeld van een dataset met twee numerieke attributen en één nominale uitkomst die prima met kleine minibatches getraind kan worden, en een andere die grote minibatches zal vereisen. Hint: de distributie van data in een minibatch dient representatief te zijn voor de distributie van data in de hele dataset; hoe hangt de haalbaarheid hiervan af van de complexiteit van de vorm van de ware scheidingsgrens tussen de twee klassen?

6.2. Learning rate decay

We hebben het zojuist al even gehad over de learning rate α. Deze belangrijke parameter bepaalt de convergentie naar een optimale oplossing. Wordt deze te groot genomen dan kan het zijn dat het model heen en weer blijft springen en nooit convergeert naar een redelijke oplossing; wordt deze te klein gekozen dan kan het heel lang duren voordat een goede oplossing gevonden wordt. Je kan dit diagnostiseren aan de hand van de validatiecurve: in het eerste geval zal de validatiecurve grillig op en neer springen; in het laatste geval zal deze slechts heel langzaam verbetering vertonen. In de praktijk kunnen beide situaties ertoe leiden dat je niet tot een geschikt model komt.

Vaak is het zinvol om aanvankelijk de learning rate behoorlijk groot te kiezen. Dit zorgt ervoor dat je vrij vlot in elk geval tot een redelijke oplossing kan komen. Daarna kan de learning rate dan worden verlaagd om te zorgen dat je niet om een oplossing heen blijft springen maar er geleidelijk naartoe convergeert. Een eenvoudige manier om hiermee om te gaan is door de learning rate handmatig bij te stellen. Je zou kunnen instellen dat je eerst een aantal epochs met hoge α traint en daarna epochs met telkens lagere waarden kiest. Je zou dit ook interactief kunnen doen door bijvoorbeeld "live" de cost van het actuele model te plotten en dan de learning rate met de hand aanpasbaar te maken. Je kan dan direct zien of het model verder verbetert door de learning rate te verkleinen.

De learning rate kan ook volgens een vast functievoorschrift geleidelijk worden verkleind. Als het aantal gedraaide epochs wordt aangeduid met t dan zijn de meest gebruikte mogelijkheden de volgende.

- Lineaire afname: $\alpha = \max(\alpha_0 - \beta \cdot t, \alpha_1)$. De waarde van α begint dan bij α_0, neemt elke epoch met een constante hoeveelheid β af, totdat een ondergrens α_1 is bereikt die dan wordt gehandhaafd.
- Exponentiële afname: $\alpha = \alpha_0 \cdot e^{-\beta \cdot t}$. De waarde begint wederom bij α_0, maar neemt elke epoch met een constante factor gerelateerd aan β af.
- Inverse afname: $\alpha = \frac{\alpha_0}{1+\beta \cdot t}$. De waarde begint bij α_0, en elke epoch wordt deze door een met β toenemend groter getal gedeeld.

Voor al deze varianten geldt dat hoe hoger β, hoe sneller de learning rate afneemt.

Tenslotte is het ook mogelijk te kiezen voor een getrapte afname. Bijvoorbeeld door elke tien epochs de learning rate te halveren, of iets dergelijks. Iets geavanceerdere methoden laten de learning rate afnemen wanneer bijvoorbeeld een minimum in de validatiecurve wordt bereikt en de cost op de validatiedata niet meer afneemt. Dat is immers het moment dat het model niet meer verder generaliseert maar dreigt te gaan overfitten.

De genoemde methoden hebben het voordeel dat ze relatief eenvoudig te implementeren zijn. Hierna zullen we echter een aantal *adaptieve* methoden bekijken die minder gevoelig zijn voor de precieze waarde van α, of die de grootte van de aanpassingen aan de modelparameters gaandeweg bepalen.

Opgave 114. *

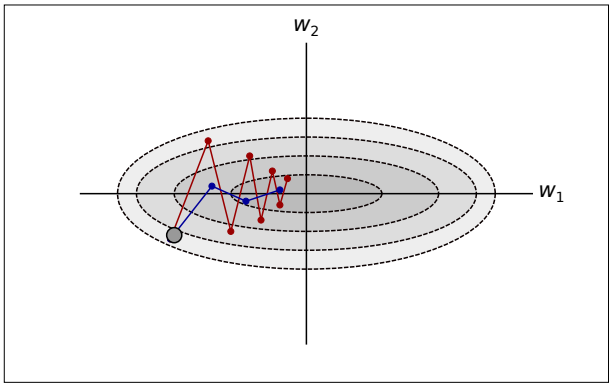
Orden de drie strategieën van afname (lineair, exponentieel, invers) van de meest langzame afname tot de meest snelle afname. Kijk hierbij naar het gedrag van de afname op de lange termijn, ongeacht de precieze waarde van de parameter β.

Opgave 115. **

Iemand suggereert de volgende strategie om de learning rate aan te passen: begonnen wordt met een zekere waarde α_0, en elke keer wanneer het teken van de verandering in een te optimaliseren modelparameter verandert (dus een positieve update in de vorige iteratie wordt gevolgd door een negatieve update in de volgende iteratie, of omgekeerd) wordt de learning rate verkleind door deze te delen door een zekere factor $\beta > 1$. Wat vind je van deze suggestie?

6.3. (Nesterov) momentum

In de onderstaande figuur staat een fenomeen geïllustreerd wat relatief vaak voorkomt. De cost J hangt hier af van twee gewichten w_1 en w_2. Dit is hier getekend in de vorm van gestippelde hoogtelijnen, zoals je die met een *contour plot* maakt. In de oorsprong van het assenstelsel behaalt de cost een minimum, aangeduid door grijstinten. Dit is in dit voorbeeld dus de optimale oplossing. Daaromheen neemt de cost geleidelijk toe. In dit geval neemt de cost sterker toe als functie van w_2 langs de y-as dan als functie van w_1 langs de x-as. De cost vormt als het ware een langgerekt dal in horizontale richting.



Gradient descent dicteert dat de stapgrootte evenredig is met de helling van de cost. In dit geval is de helling langs de horizontale richting kleiner dan die langs de vertikale richting. De stappen waarmee de gewichten worden bijgewerkt zullen dus ook groter zijn voor w_2 dan voor w_1. De gradiënt staat altijd exact loodrecht op de hoogtelijnen en is groter naarmate de hoogtelijnen dichter op elkaar lopen. Hetzelfde geldt voor de stappen.

Stel, de gewichten hebben op een zeker moment waarden die overeenkomen met de grijze stip. Dan zal, om de helling af te gaan, dus een stap gezet worden die groot is naar boven en klein is naar rechts. Voor de volgende stappen geldt elke keer iets vergelijkbaars. Deze volgen dan ongeveer de rode zigzaglijn. In deze illustratie kom je telkens helemaal aan de andere kant van het dal terecht en ben je tegelijkertijd een beetje opgeschoven naar rechts. In dit geval nadert de serie rode stappen volgens gradient descent weliswaar het optimum, maar de zwalkende manier waarop is verre van ideaal.

Afhankelijk van de grootte van de learning rate α kan de stapgrootte iets te groot zijn waardoor je over de ideale waarde heenschiet of te klein waardoor je niet ver genoeg komt. Wat het geval is kan echter per modelparameter verschillen, zien we nu. Zo is hierboven voor w_1 de learning rate eigenlijk te klein, want je gaat niet zover naar rechts als je wel zou willen om in het optimum te belanden. Maar tegelijkertijd is voor w_2 de learning rate te groot, want je schiet in vertikale richting helemaal door het dal heen om aan de andere kant van het laagste punt te belanden. Het is dus onmogelijk een learning rate te kiezen die alle parameters even goed bijwerkt.

Een manier om dit gedrag te verbeteren is door niet alleen te kijken naar wat de gradiënt op een zeker moment is, maar ook naar wat de gradiënt de afgelopen stappen was. Zou je bijvoorbeeld naar het gemiddelde kijken over meerdere stappen, dan krijg je meteen al een veel beter gedrag. Immers, de gradiënten in de vertikale richting wisselen steeds af van teken. Als je die middelt dan werken ze tegen elkaar in en blijft er een kleine stapgrootte over. De gradiënten in de horizontale richting daarentegen wijzen allemaal dezelfde kant op. Als je deze middelt dan blijven ze dus aanzienlijk. Als je stappen zou zetten die gelijk zijn aan het gemiddelde van de afgelopen gradiënten vermindert het zig-zag patroon sterk. Je kan dan veel sneller convergeren naar het optimum, zoals getoond met de blauwe lijn.

Je kan dit voorstellen als een zekere massa die aan de stip wordt gehangen. Als die eenmaal in beweging is verandert die niet zomaar weer. In vertikale richting werkt

de gradiëntkracht op het punt steeds andersom, waardoor de beweging uitmiddelt. In horizontale richting werkt de gradiëntkracht telkens dezelfde kant op, waardoor deze beweging relatief wordt versterkt. De uitmiddelende werking van een dergelijke massa wordt ook wel *momentum* genoemd.

De traditionele manier waarop gradient descent werkt is dat er voor elke parameter w een gradiënt $G = \frac{\partial J}{\partial w}$ wordt bepaald en dat vervolgens de parameter wordt bijgewerkt volgens de regel

$$w \leftarrow w - \alpha G$$

Dit is nog steeds gewoon de updateregel volgens gradient descent, alleen slaan we de gradiënt nu op in een speciale variabele G. Wanneer we gebruik maken van momentum moeten we niet bijwerken met de gradiënt, maar met een gewogen voortschrijdend gemiddelde of *moving average* daarvan. We bereiken dit door G zelf geleidelijk bij te werken volgens

$$G \leftarrow \beta G + \frac{\partial J}{\partial w}$$

De coëfficiënt $0 < \beta < 1$ bepaalt in dit geval hoe lang de eerdere gradiënten blijven meewegen. Dit komt als het ware overeen met de massa van de zwarte stip. Kiezen we β nabij 0, dan krijgen we precies onze "oude" gradient descent updateregel terug. De massa is dan nul. Nadert β naar 1 dan wordt G steeds sterker bepaald door de voorafgaande gradiënten in plaats van door de huidige gradiënt $\frac{\partial J}{\partial w}$. Dit helpt om een steeds grotere snelheid in de juiste richting op te bouwen. De massa is dan hoog. De coëfficiënt β wordt ook wel de *momentum parameter* of *frictie parameter* genoemd. De term frictie duidt erop dat kleine waarden van β een afremmende werking hebben en lijken op wrijving. Een typische waarde voor β is 0.9. Dit betekent losjes gesproken dat je de stap voor 90% laat afhangen van de voorafgaande gradiënten en voor de resterende 10% van de huidige nieuwe gradiënt.

Met behulp van momentum kan het leerproces versnellen omdat de voorkeur wordt gegeven aan richtingen die consistent zijn over meerdere iteraties heen, terwijl nutteloze zijwaartse oscillaties worden gedempt. Het leidt er echter ook toe dat je over de optimale oplossing heen kan schieten zodra je die bereikt hebt, omdat de opgebouwde snelheid blijft doorwerken. Vergelijk dit met een knikker in een ronde kom die naar beneden rolt en dan doorschiet. Echter, de winst die je kan bereiken door sneller naar het optimum te convergeren compenseert gewoonlijk ruimschoots voor de extra tijd die je nodig hebt om rond het optimum tot rust te komen. Het hebben van een zekere *overshoot* kan zelfs gunstige gevolgen hebben als het bereikte minimum slechts een lokaal minimum blijkt te zijn: de opgebouwde impuls kan dan voldoende zijn om door het lokale minimum heen te schieten om daarna verderop wellicht in een beter minimum te belanden.

Het gebruik van momentum heeft nog een extra voordeel. Omdat je continu een schatting G bijhoudt van de gradiënt en die geleidelijk bijwerkt, weet je tijdens de forward-propagation fase al redelijk hoe de nieuwe gewichten er uit gaan zien, zelfs als die nog niet zijn bijgewerkt voor de lopende minibatch. Immers, je weet dat je de updateregel $w \leftarrow w - \alpha G$ zal gaan toepassen, en hoewel je nog niet exact weet wat de waarde van de nieuwe gradiënt G gaat zijn, is de huidige waarde daarvan al best een aardige

schatting. Je kunt dus tijdens de forward-propagation alvast een voorschot nemen op de update, en werken met de gewichten zoals je die zou gaan krijgen met de huidige waarde van G. Vervolgens bereken je de gradiënt $\frac{\partial J}{\partial w}$ voor de huidige minibatch, uitgaande van die schatting van de nieuwe gewichten. Daarmee werk je dan de gradiënt bij volgens $G \leftarrow \beta G + \frac{\partial J}{\partial w}$, en pas dan update je de gewichten definitief met behulp van die nieuwe waarde van G. Het gevolg is dat je de gewichten kan bijwerken met een nauwkeurigere nieuwe schatting van de gradiënt. Je hebt als het ware al vooruit kunnen kijken naar waar je terecht gaat komen. Hierdoor kun je bijvoorbeeld al beginnen "af te remmen" nog vóórdat je in een minimum bent beland, waardoor je er dus minder sterk doorheen zal schieten. Het idee om tijdens de forward propagation alvast gebruik te maken van de bestaande gemiddelde gradiënt om een betere schatting te krijgen van de nieuwe gradiënt wordt *Nesterov momentum* of *Nesterov accelerated gradient* genoemd.

Opgave 116. *

Er werd gezegd dat een voordeel van momentum is dat je door een lokaal minimum heen kan rollen om elders in een dieper minimum te belanden. Echter, omgekeerd zou je natuurlijk ook door een globaal minimum heen kunnen rollen en in een ondieper minimum kunnen blijven steken. Bedenk waarom het netto meestal toch voordelig uitwerkt.

Opgave 117. **

Leg uit waarom het idee van Nesterov om vooruit te kijken alleen gebruikt kan worden als je gebruik maakt van momentum, maar niet als je gebruik maakt van "gewone" (stochastic) gradient descent.

Opgave 118. ***

Stel, de update zou worden uitgevoerd volgens $G \leftarrow \beta G + (1 - \beta) \frac{\partial J}{\partial w}$? Wat is hiervan de invloed op de werking van het algoritme? Hoe zou je de waarde van α kunnen aanpassen zodat de uitkomsten identiek zouden worden als bij de update regels eerder hierboven?

6.4. Gradiënten schalen

In de vorige paragraaf werd beschreven hoe door de gradiënten te middelen over meerdere iteraties de convergentie kon worden verbeterd. Dit doet niet af aan de mogelijk langzame convergentie die kan optreden wanneer de activatie- of lossfuncties dreigen te satureren. In hoofdstuk 4 hebben we immers gezien dat weliswaar de saturatie van de softmax-functie kan worden tegengegaan door een cross-entropy lossfunctie te gebruiken, maar saturerende activatiefuncties in eerdere hidden layers kunnen nog steeds optreden en leiden tot stagnerende leerprestaties van het neurale netwerk. Het kan in zo'n geval gebeuren dat een parameter weliswaar consistente stappen maakt in de juiste richting, maar dat die stappen slechts heel klein zijn. Het middelen van gradiënten met momentum werkt dan niet: het gemiddelde van zwakke gradiënten blijft een zwakke gradiënt. Een ander idee is daarom om de gradiënt te schalen. Als de grootte van de gradiënt systematisch klein dreigt te worden dienen we de stapgrootte op te schalen.

Een methode die dit idee handen en voeten geeft heet *RMSProp*, wat staat voor *root-mean-square propagation*. Hierbij wordt een variabele bijgehouden die een idee geeft van de grootte van de gradiënt en die wij hier Q zullen noemen. Dit wordt bereikt door een lopend gemiddelde te nemen van de kwadraten van de gradiënt, volgens

$$Q \leftarrow \beta Q + (1-\beta)\left(\frac{\partial J}{\partial w}\right)^2$$

Hier komen waarden β en $(1-\beta)$ tussen 0 en 1 in voor die zorgen dat voorafgaande iteraties weliswaar worden meegewogen maar ook weer geleidelijk worden vergeten. Opnieuw geeft de *decay parameter* β aan hoe sterk voorafgaande iteraties meetellen. Dit is vergelijkbaar met momentum, zij het dat niet de gradiënt wordt uitgemiddeld maar diens kwadraat.

Hoe groter de gradiënten zijn, ongeacht hun teken, hoe sterker de waarde van Q zal oplopen. Q geeft dus een idee van de sterkte van alle gradiënten tot nu toe, zonder naar hun richting te kijken. De formule die wordt gehanteerd om de modelparameters bij te werken luidt vervolgens

$$w \leftarrow w - \frac{\alpha}{\sqrt{Q+\epsilon}}\frac{\partial J}{\partial w}$$

Hierbij is in de noemer de schaalfactor ongeveer gelijk gekozen aan $\sqrt{Q}$. Immers, Q houdt de kwadraten van de gradiënt bij, dus om iets over de grootte van de gradiënt zelf te kunnen zeggen dien je hiervan de wortel te nemen. Verder is er een constante ϵ toegevoegd om problemen met delen door nul te vermijden. Deze wordt uiterst klein gekozen, bijvoorbeeld 10^{-6} of 10^{-9}, zodat deze meestal geen merkbare invloed heeft.

Als er nu bijvoorbeeld al vrij snel saturatie optreedt en de gradiënt blijft maar klein voor een bepaalde parameter, dan zal ook $\sqrt{Q}$ een betrekkelijk kleine waarde houden. In de updateregel wordt gedeeld door deze kleine waarde, dus dat betekent dat de stapgrootte in dat geval naar verhouding groot kan blijven. Het effect van de bovenstaande formule is daardoor dat het er voor de stapgroottes eigenlijk niet meer zoveel toe doet hoe groot de gradiënten precies zijn: gekeken wordt vooral naar de grootte van de gradiënt in verhouding tot de gradiënten zoals die tot dan toe zijn opgetreden. Hoe groter die verhouding, hoe groter de stappen. Dat betekent dat een modelparameter die satureert en daardoor nog slechts hele kleine stapjes dreigt te maken daardoor minder wordt beïnvloed. Immers, weliswaar wordt de gradiënt dan klein, maar omdat de gradiënt dan ook in het verleden klein was kan in verhouding toch een grote stap worden gezet.

Opgave 119. **

Is het ook redelijk om de update uit te voeren volgens $w \leftarrow w - \frac{\alpha}{\epsilon+\sqrt{Q}}\frac{\partial J}{\partial w}$, denk je? (Let op de verschillende positie van de constante ϵ.) Waarom werkt dit wel/niet?

Opgave 120. **

Stel dat voor een bepaalde modelparameter de grootte en richting van de gradiënt constant zijn. Met andere woorden, in alle iteraties blijkt de gradiënt $\frac{\partial J}{\partial w}$ exact dezelfde

waarde te hebben. Hoe gedraagt zich dan op de lange termijn de stapgrootte volgens de updateregel van RMSProp? Neigt die naar een constante, en zo ja hoe groot is dan die constante, of wordt die steeds groter of kleiner? Wat vertelt je dit over de gevoeligheid voor saturatie?

6.5. Adaptive moments

Tenslotte kunnen de ideeën van momentum en RMSProp worden gecombineerd door zowel een lopend gemiddelde voor de gradiënt als voor diens kwadraat bij te houden. We krijgen dan een lopend gemiddelde voor de gradiënt volgens

$$G \leftarrow \beta_1 G + (1 - \beta_1) \frac{\partial J}{\partial w}$$

en een lopend gemiddelde voor het kwadraat van de gradiënt volgens

$$Q \leftarrow \beta_2 Q + (1 - \beta_2) \left(\frac{\partial J}{\partial w} \right)^2$$

De updateregel wordt daarmee

$$w \leftarrow w - \frac{\alpha G}{\sqrt{Q + \epsilon}}$$

De twee decay parameters β_1 en β_2 geven aan hoe zwaar voorafgaande iteraties worden meegewogen voor respectievelijk de gradiënt zelf en diens kwadraat. Deze hoeven niet noodzakelijk gelijk aan elkaar gekozen te worden. In de praktijk wordt bijvoorbeeld vaak $\beta_1 = 0.9$ en $\beta_2 = 0.999$ genomen. Dit betekent dat de richting van de gradiënt in staat is om zich sneller aan te passen dan de schaling met de diens grootte.

Een probleem dat bij deze methode kan optreden is gerelateerd aan de initialisatie van de parameters G en Q. Die worden normaal aanvankelijk gelijk aan nul gesteld. Maar dat betekent dat in de eerste paar updates de gradiënt te klein of te groot genomen wordt omdat de nulwaarde nog even door blijft werken. Er wordt dan ook wel gesproken van een *bias*: de gemiddelde gradiënt en diens grootte worden aanvankelijk beide systematisch onderschat. Dit kan met name vervelende gevolgen hebben voor de bovenstaande methode omdat hierin G en Q deze nulwaarde met een ander tempo laten uitdoven. Om dit te voorkomen kun je *unbiased* waarden voor G en Q berekenen volgens

$$G' = \frac{G}{1 - \beta_1^t}$$

en

$$Q' = \frac{Q}{1 - \beta_2^t}$$

In de noemer worden de wegingsfactoren tot de t-de macht verheven, waarbij t het aantal uitgevoerde iteraties is, dat wil zeggen het aantal minibatches. Bij de eerste iteratie is $t = 1$. Deze correctie heft precies het effect van de aanvankelijke nulwaarden op.

In de updateregel moeten dan natuurlijk deze unbiased waarden worden verwerkt

$$w \leftarrow w - \frac{\alpha G'}{\sqrt{Q' + \epsilon}}$$

De methode die nu is verkregen staat bekend als *Adam*, wat staat voor *adaptive moments estimation*. Deze methode is zeer populair omdat ie eigenlijk alle ingrediënten van van de eerdere methoden in zich bergt, en tegelijkertijd toch nog (betrekkelijk) eenvoudig te implementeren is.

Het is tenslotte mogelijk om hier ook nog het idee van Nesterov momentum aan toe te voegen, waarbij je tijdens de forward-propagation alvast een voorschot neemt op de update die je zal gaan maken in de gewichten. Immers, ook bij Adam houd je waarden voor G en Q bij die slechts langzaam zullen veranderen en dus redelijk voorspelbaar zijn. Het resultaat wordt *Nadam* genoemd, maar deze methode lijkt niet populairder te zijn dan Adam zelf.

Overigens bestaan er nog diverse andere varianten. Het voert te ver om die hier allemaal uiteen te zetten.

Opgave 121. **

Als je de beide frictieparameters gelijk aan nul stelt, zodat $\beta_1 = 0$ en $\beta_2 = 0$, hoe groot is dan in elke iteratie de stapgrootte volgens Adam? Je mag de constante ϵ verwaarlozen.

Opgave 122. **

Stel, je kiest $\beta_1 = \beta_2$ bij Adam. Zal dan de waarde van Q gelijk zijn en blijven aan het kwadraat van G? Oftewel, is een (gewogen) gemiddelde van kwadraten hetzelfde als het kwadraat van een (gewogen) gemiddelde?

Opgave 123. **

Toon aan dat ongeacht de precieze vorm van de cost na de eerste iteratie $G' = \frac{\partial J}{\partial w}$ en $Q' = \left(\frac{\partial J}{\partial w}\right)^2$. Aannemende dat de constante ϵ verwaarloosbaar klein is, hoe groot is dan de verandering in de modelparameter w?

Opgave 124. **

Hoe moeten de parameters β_1 en β_2 gekozen worden om Adam te reduceren tot RMSProp?

Opgave 125. ***

Leg uit dat je zowel een bias kunt hebben richting te grote updates als richting te kleine updates. Hoe hangt dit af van de parameters β_1 en β_2?

Opgave 126. ***

Is het mogelijk om met een geschikte keuze van de parameters β_1 en β_2 hetzelfde gedrag te krijgen als met momentum alleen? Leg je antwoord uit.