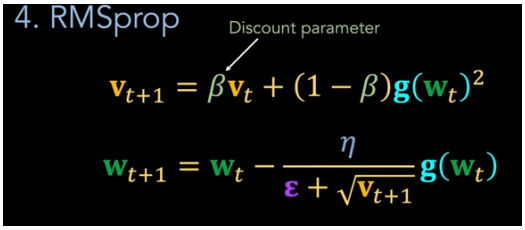
تمرین تئوری سری دوم شبکه عصبی

سارا حسینی ۴۰۰۲۲۲۰۲۶

۱

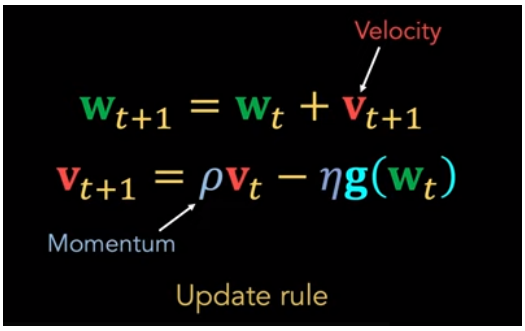
مشتق تابع فعالیت در محل pre-activation های متفاوت، حین محاسبه‌ی Δ (سهم خطای هر نورون)، به صورت زنجیری در هم ضرب می‌شود. همین باعث میشود که گرادیان هر چه عقبتر برود کوچک و کوچکتر شود. مثلا سیگموید چون مشتق بسیار کوچکی دارد (ماکسیمم 0.25) در حین بکپروپ دلتاها را کم و کمتر میکند. این مشکل را vanish شدن گرادیان میگویند.
از طرفی اگر خود وزن‌ها هم از order بالایی باشند مثلا بزرگتر از ۱۰، دلتا در حین بکپروپ بسیار بزرگ میشود. همچنین مشتق تابع فعالیت هم میتواند بزرگ باشد مثلا اگر از ReLU استفاده کنیم در مقادیر مثبت مشتق همواره یک است و باعث بزرگی دلتا خواهد شد که به آن explode شدن گرادیان میگویند.

در واقع جهت گرادیان همواره درست است اما اندازه آن میتواند بسیار بزرگ شود که موجب جهش‌های بزرگ در وزن‌ها و عدم یافتن اپتیمم شود، یا بسیار کوچک شود که سبب کوچکی مقدار تغییر وزن‌ها و از کار افتادن شبکه شود.
یک روش برای جلوگیری از جهش‌ها در وزن، استفاده از learning rate مناسب است که مسیر را هموارتر کند. learning rate های ثابت، میتوانند باعث ناپایداری گرادیان شوند. مثلا اگر کوچک باشد، سبب vanish و اگر بزرگ باشد، سبب explode شدن گرادیان میشوند. پس روشهایی که با آن میتوان learning rate را داینامیک و آداپتیو کرد، به ما کمک میکنند. مومنتوم و RMSProp از روشهای آداپتیو کردن learning rate هستند. در RMSProp هر چه تغییر قبلی یک وزن، کوچکتر باشد، تغییر فعلی آن بزرگتر میشود. یعنی اگر اخیرا زیاد تغییر نکرده باشد، حالا به آن اجازه تغییر بیشتر میدهد. اگر هم اخیرا زیاد تغییر کرده باشد، کمتر به آن اجازه تغییر میدهیم.



شکل ۱: هر چه β کوچکتر شود، بطور محلی‌تری وزن را آپدیت میکنیم. هرچه β بزرگتر شود، تاثیر گرادیان‌های پیشین بیشتر میشود.

در مومنتوم هم نوعی حافظه به وزن میدهیم که گرادیان‌های اخیر را هم در آپدیت وزن، تاثیر دهد.



شکل ۲: Momentum learning rule

آدام، ترکیب مومنتوم و RMSProp است و از هر دویشان قوی‌تر است. اندازه گرادیان فعلی، با توجه به مربع گرادیان‌های اخیر scale میشود و همچنین با مومنتوم، بطور هموار به اپتیمم حرکت میکنیم.
تابع فعالیت ReLU میتواند سبب مشکل "مردن" نورون‌ها شود یعنی برای مقادیر منفی که گرادیان صفر است، این نورون‌ها ممکن است کلا از کار بیفتند. اما تابع leaky ReLU این مشکل را حل میکند چون در مقادیر منفی مقدلر کوچکی مشتق دارد که میتواند امکان ریکاوری و آپدیت دوباره دهد و از مرگ نورون جلوگیری کند.

۲

■۱- فاز آموزش را طولانی نکنیم تا مدل بیش از حد یاد نگیرد و شروع به حفظ کردن نکند. به این روش early stopping میگوییم. برای اینکه تشخیص دهیم چه زمان باید متوقف شویم، باید از validation set استفاده کنیم. شبکه را با داده‌ی آموزشی train میکنیم. بعد از هر epoch ارور را هم روی داده‌ی آموزشی و هم روی ولیدیشن، حساب میکنیم. به همین صورت ادامه میدهیم.

جایی میرسد که ارور برای داده ترین در حال کاهش است اما ناگهان روی داده ولیدیشن، ارور افزایش می‌یابد. این یعنی مدل در آستانه اورفیت شدن است. باید تا تعداد epoch موردنظر، آموزش را انجام دهیم و سپس تعداد epoch ای که کمترین ارور را روی ولیدیشن داده بود، بعنوان بهترین تعداد تعداد epoch در نظر میگیریم و سپس دوباره با آن تعداد مدل جدید را آموزش میدهیم. یا میتوانیم بعد از هر epoch اگر ارور روی ولیدیشن مینیمم بود، وزن‌ها را سیو کنیم.چون ممکن است منحنی تابع هزینه هموار نباشد و افزایش سپس کاهش بیشتری داشته باشد پس باید به این نکته توجه کنیم و میتوانیم از تکنیک‌های هموارسازی مثل میانگین متحرک هم استفاده کنیم.

■۲- feature selection. اگر تعداد ستون‌های زیادی داشته باشیم، احتمالا بخشی از آنان بی‌اهمیت هستند یا به پیش‌بینی ما کمکی نمیکنند. با دراپ کردن این ستون‌ها و انتخاب فیچرهای مهم، پیچیدگی مدل ما کاهش می‌یابد و احتمال اورفیت کمتر میشود.

■۳-شبکه را کوچک کنیم. نورون‌هایش را کم کنیم تا پیچیدگی آن پایین بیابد. البته این روش مورد مناسبی نیست چون شاید از پیچیدگی مسئله کمتر بشود.

■۴-از تکنیک‌های رگولاریزیشن استفاده کنیم تا پیچیدگی مدل بطور داینامیک تنظیم شود. مدلی با تعداد زیاد پارامتر، باعث اورفیت میشود. عبارات رگولاریزیشن باعث میشوند شبکه تلاش کند با استفاده از کمترین تعداد نورون، مسئله را یاد بگیرد. بدین صورت که رگولاریزیشن مدل را مجبور میکند اندازه نرم وزن‌ها را پایین بیاورد و در عین حال ارور آنقدر بالا نرود. برای این کار میتوان از L_1 (نرم ۱ وزن‌ها)، L_2 (نرم ۲ وزن‌ها) یا ترم‌های پنالتی دیگر، مثل مجموع خروجی‌های تمام لایه‌ها استفاده کرد.

■ ۵- استفاده از متدهای انسامبل. یعنی بجای یک مدل قوی، چندین مدل نسبتا قوی داشته باشیم که با هم متفاوتند. این مدل‌ها چون ضعیفترند، احتمال اورفیت شدنشان پایین است.

■۶- روش dropout. یعنی از درصدی از نورونها که رندوم انتخاب شده‌اند، برای سمپل جاری استفاده نکنیم. دو رویکرد در اینجا داریم: اول) در مسیربرگشت نورونها را دراپ کنیم. یعنی اینکه اصلاح وزن توسط بکپروپ را فقط روی یک بخشی از نورون‌ها اعمال کنیم. (انگار داریم شبکه‌هایی ضعیفتر را در دل شبکه قوی خودمان ترین میکنیم). دوم)هم در مسیر رفت هم برگشت، نورونها را دراپ کنیم.

■۷- روش dropconnect. مثل بالایی اما بجای نورون، وزن‌ها/کانکشن‌ها را دراپ میکنیم و فرض میکنیم این وزن‌ها صفرند.

■۸- افزایش دادن تعداد داده‌ی آموزشی. با این کار سعی ما این است که پیچیدگی دیتا را بالا ببریم تا به پیچیدگی مسئله برسد. اما این کار ممکن است سخت یا پرهزینه باشد.

■۹- میتوانیم از روش‌های augment کردن دیتا استفاده کنیم مثلا میرور کردن، روتیت کردن، کراپ کردن عکس، اعوجاج افزودن به عکس، دستکاری فرکانس‌های دیتای صورتی. یا میتوانیم از مدل‌های جنریتیو برای تولید دیتا استفاده کنیم.

۳

بله. تکنیک دراپ‌اوت با حذف نورون‌ها، عملا شبکه را ساده‌تر می‌کند پس شبکه به زمان بیشتری برای همگرایی به اپتیمم نیاز خواهد داشت. یعنی با تعداد epoch های بیشتری ما را به اپتیمم میرساند. همچنین در حین محاسبه، به یک ماتریس جدا برای mask کردن وزن‌ها متصب به نورون خاموش‌شده نیاز داریم که به حجم محاسبات می‌افزاید. به این مسک، dropu-out layer هم میگوییم. همچنین هر بار (با هر سمپل) برای رندوم انتخاب کردن نورون‌هایی که باید صفر شوند، نیاز است که یک عملیات نمونه‌گیری از اعداد صورت گیرد. دراپ‌اوت باعث میشود یک سطح از رندوم بودن به شبکه اضافه شود و احتمال اینکه در جهات اشتباهی حرکت کنیم و در مسیر از اپتیمم دور شویم و دوباره برگردیم بالا برود.
بله ممکن است کمی زمان پیشبینی هم افزاش یابد، چون یک سری نورون در حین آموزش، خاموش بوده‌اند، باید حین تست/پیش‌بینی، روشن باشند. پس باید یک ضرب اضافه انجام دهیم تا محاسبات به هم نریزد. البته این قضیه را هنگام آموزش، با ضرب همه وزن‌های لایه‌هایی که دراپ‌اوت داشته‌اند، در $1 - dropoutrate$ میتوان حل کرد. اما بجز این، محاسبه اضافه‌ای در حین تست ندارد و تاثیر زیادی بر سرعت پیش‌بینی نمیگذارد.
Monte Carlo dropout به این صورت است که حین تست هم، dropout layer ها فعالند. برای هر تست، n بار، آن را از شبکه عبور میدهیم و هر بار نورون‌های مختلفی deactivated میشوند پس ۱۰ جواب مختلف میگیریم. سپس میانگین n جواب را میگیریم و بعنوان جواب نهایی معرفی میکنیم. ر این حالت، زمان موردنیاز برای پیشبینی بالاتر از حالت نورمال میرود.

۴

L_1 و L_2 هر دو ترم پنالتی هستند که وقتی به loss function اضافه میشوند، سعی میکنند مدل را مجبور کنند اندازه وزنها را نزدیک به صفر نگه دارد. و از وزن‌های بسیار بزرگ دوری کند ولی ارور را هم کوچک نگه دارد. نورم ۱ سختگیرانه‌تر است و وزن‌های زیادتری را صفر میکند و مدل را sparse میکند . چون در L_2 وزن‌های کوچک وقتی به توان میرسند (مثلا وزن‌های بین ۰ و ۱) کوچکتر میشوند ولی وزن‌های بزرگ، بزرگتر میشوند. یعنی مدل به وزن‌های کوچک اهمیت نمیدهد و بیشتر سعی دارد آنهایی که خیلی بزرگند را کوچک کند. ولی L_1 همه‌ی وزن‌ها را. وقتی با دیتایی که ابعاد زیادی دارد سر و کار داریم، میتوانیم از L_1 استفاده کنیم چون باعث میشود مدل sparse تر شود و فقط فیچرهای مهم را یاد بگیرد و قابلیت generalize مدل بالا میرود. هرچه λ بزرگتر شود، وزن‌های بیشتری صفر میشوند و بیشتر رگولاریزیشن اعمال میشود:

$$\text{Loss} = \tfrac{1}{2}\Sigma(y - \hat{y})^2 + \lambda\frac{\sum_i |w_i|}{2}$$

در L_2 داریم: $\text{Loss} = \tfrac{1}{2}\Sigma(y - \hat{y})^2 + \lambda\frac{\sum_i w_i^2}{2}$

این یعنی اینکه ال۲ میتواند به outlier ها ارور بالا بدهد و آنها را بهتر penalize کند. ال ۱ میتواند برخی وزن‌ها را کاملا صفر کند اما ال۲ آنها را نزدیک صفر میکند. چون مثلا وقتی اندازه وزن 0.1 باشد، ال ۱ به ما همان 0.1 را میدهد ولی ال ۲ 0.01 میدهد. یعنی ده برابر کوچکتر. پس همین وزن ما به مقدار کوچکی مثل 0.1 رسیید، ال۲ خیلی این وزن را نزدیکتر به اپتیمم میبیند تا ال۱ و نیازی نمیبیند خیلی برای صفر کردنش تلاش کند (به اندازه ال۱).

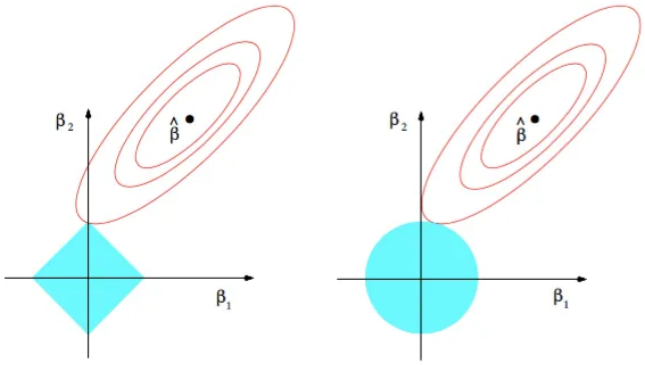
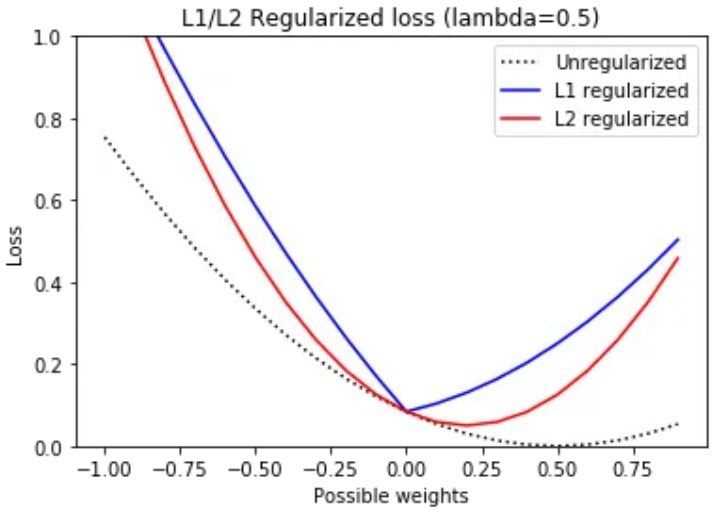


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

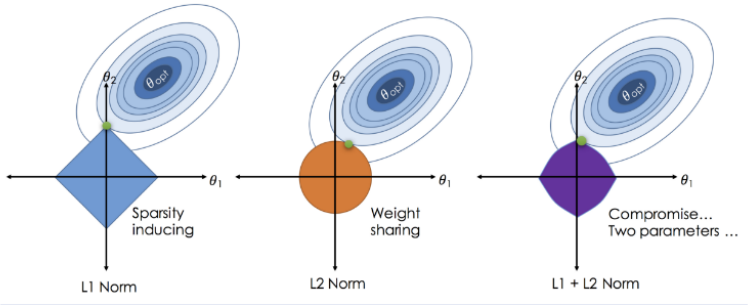
شکل ۳: سمت راست L_2 و سمت چپ L_1. نشان میدهد چرا L_1 وزن‌ها را صفر میکند ولی L_2 نه. این شکل برای مدلی با دو وزن کشیده شده



شکل ۴: این شکل تفاوت را با یک وزن نشان میدهد

فرق دیگر است است که ال۱ نسبت به فیچرهای correlated حساسیت کمتری دارد. چون، کانتور پلات ارور معمولا در نقطه‌ای از گوشه‌های مربع ال۱، آن را قطع میکند. پس تعدادی فیچر صفر میشود و از میان تعدادی فیچر که به هم وابسته correlated هستند، یکی انتخاب میشود.
الستیک نت، چیزی بین ال۱ و ال۲ است. از ال۱ برای فیچر سلکشن و صفر کردن بعضی وزن‌ها، و از ال۲ برای انتخاب فیچرهای بیشتر استفاده میکند.

$$\text{Loss} = \tfrac{1}{2}\Sigma(y-\hat{y})^2 + \lambda_1 \frac{\Sigma_i |w_i|}{2} + \lambda_2 \frac{\Sigma_i w_i^2}{2}$$



الستیک نت، نسبت به ال۱ یا ال۲، robust تر است. برای وقتی که تعداد فیچرهای زیاد در دیتا داریم ولی تعداد نمونه‌های دیتا کمتر است، مناسب است.

۵

batch normalization باعث میشود سرعت یادگیری بالا برود و زودتر به اپتیمم برسیم. با این فرمول، باید روی ورودی تمام لایه‌ها، نورمالیزیشن را اعمال کنیم: (دقت کنید هر نورون با هر بچ، میانگین و واریانس خودش را دارد)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

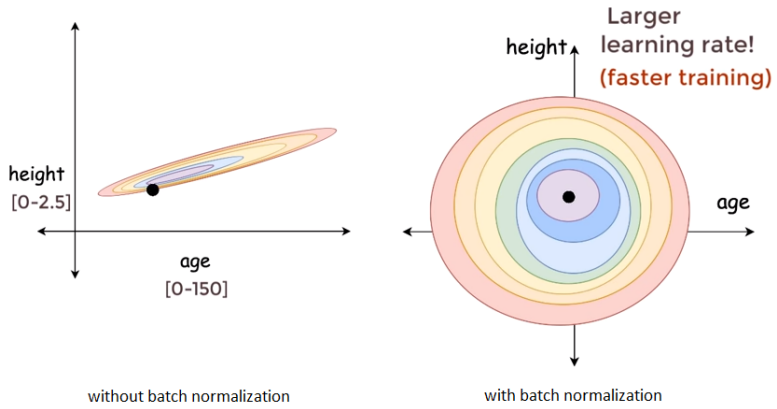
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.



شکل ۵: سمت راست با نورمالیزیشن و چپ بدون نورمالیزیشن

همانطور که در شکل میبینیم، اگر ورودی‌ها نورمال نباشند، تابع هزینه (کانتور پلات) همواری یکسانی در همه جا ندارد. یعنی لرنینگ ریت باید بسیار کم باشد تا بتوانیم بفهمیم برای متغیر height در کدام جهت باید حرکت کنیم چون تغییرات کوچک در این متغیر، هزینه را بسیار زیاد تغییر میدهند (overshoot اتفاق میفتد). اما وقتی نورمالایز شده باشد، سمت راست، میتوان با لرنینگ ریت بالا حرکت کرد و به مینیمم رسید چون تابع هموار و متقارن و evenly spread است. (یک راه دیگر برای اینکه این قضیه را هندل کنیم استفاده از لرنینگ ریت آدام است که به هر متغیر یک ریت جدا نسبت میدهد) همچنین وقتی نورمالایز نکرده ایم، اگر جایی در قسمت‌هایی که خطوط کانتور از هم دورترند یعنی تابع شیب کمتر دارد، شروع کنیم چندین برابر iteration بیشتری باید انجام دهیم تا به مینیمم برسیم نسبت به وقتی که در نواحی با خطوط نزدیک یعنی شیب تندتر شروع کنیم. اما وقتی نورمالایز کرده‌ایم، اهمیتی ندارد از کجا شروع کنیم. از هر جا شروع کنیم در تعداد iteration با اردر یکسانی به مینیمم میرسیم. این یعنی با نورمال سازی، دیگر وزن‌دهی اولیه در سرعت همگرایی آنقدر موثر نیست.

۶

یک ensemble از چند مدل، یعنی دیتا به چند مدل داده شود و آن مدلها هر کدام یک جواب و نهایتا بر اساس جوابهای آنها جواب نهایی داده شود. مدل‌های ضعیف، نواقص همدیگر را پوشش میدهند و کنار هم یک مدل انسامبل قوی میسازند.

دو دسته مدل انسامبل داریم: homogeneous که یعنی مدل‌های ضعیف، از یک جنس هستند مثلا همگی درخت تصمیم هستند و heterogeneous که در آن مدل‌های ضعیف متفاوت هستند. روش بگینگ و بوستینگ با مدل‌های homogeneous و روش استکینگ با مدل‌های

heterogeneous ساخته میشوند. در بگینگ واریانس مدل‌های ضعیف را کم میکنیم و این مدلها مستقل از همند. دیتای آموزشی را با روش bootstrap نمونه‌گیری میکنیم و هر مدل را با یکی از نمونه‌ها اموزش میدهیم. سپس با روش میانگین‌گیری یا رای‌دهی برای کلسیفیکیشن، نتیجه را اعلام میکنیم. در بوستینگ، مدلهای ضعیف به طور متوالی به هم متصل میشوند. سپس بعد از آنکه مدل اول جوابش را داد، آن دیتاهایی که اشتباه کلسیفای کرده بود را در دیتاست افزایش میدهیم و این دیتاست اپدیت شده را به مدل بعدی میدهیم تا بتواند اشتباهات مدل اول را یاد بگیرد و کاور کند. برای ترکیب نتایج مدل، یکی از راه‌ها استفاده از مجموع وزن‌دار است.

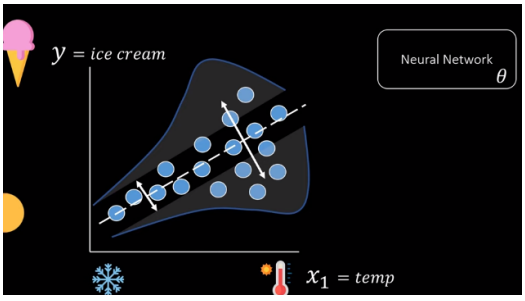
$$f_{ensemble}(x) = \sum_i c_i f_i(x)$$

که c_i وزن مرتبط به هر مدل و $f_i(x)$ خود مدل است.
روش استکینگ نیز بدین صورت است که دیتا را k قسمت میکنیم سپس k-1 تا را برای آموزش به تمام مدلهای ضعیف میدهیم.بعد یک قسمت باقیمانده را به مدل‌ها میدهیم تا روی آن به ما جواب بدهند. آنگاه با استفاده از جواب‌های آنها، یک مدل ثانویه را آموزش میدهیم تا بتواند از روی آین جوابها، جواب نهایی را بفهمد.
مدل‌های شبکه عصبی، واریانس بالا و بایس پایینی دارند یعنی ممکن است اورفیت شوند یا با هر بار ترین شدن، نتایج مختلفی بدست بدهند و mapping متفاوتی از ورودی به خروجی را یاد بگیرند. برای اینکه با این واریانس بالا مقابله کنیم میتوانیم چندین مدل را با یک دیتا آموزش دهیم و پیشبینیهایان را ترکیب کنیم. این مدل ها باید خوب ولی متفاوت باشند یعنی خطاهایشان در جاهای مختلفی اتفاق بیفتد. یکی از راه‌های مهم این کار به صورت committee of networks است. چند شبکه با ساختار یکسان ولی وزن‌های اولیه متفاوت روی دیتای یکسان اموزش میگیرند. سپس پیشبینی مدل‌ها، میانگین‌گیری میشود. معمولا تعداد مدل‌ها زیر ۱۰ است چون بیشتر باعث محاسبات بسیار زیاد و هزینه‌بر میشود.
در نهایت، مدل‌های انسامبل مختلف را با تغییر هر یک از این سه عامل میتوان بدست آورد: "روش ترکیب پیشبینی مدل‌ها"، یا "خود مدل‌های ضعیفتر" یا "دیتایی که به مدلها میدهیم".
برای انتخاب دیتا، میتوان دیتا را به چند قسمت تقسیم کرد و هر کدام را به یک مدل داد (k-fold cross-validation) یا از طریق resampling برای هر مدل، نمونه ای از دیتا را مشخص میکنیم که این همان بگینگ است.
در مورد مدلها، همیشه ممکن است مدلها رو یک لوکال مینیمم گیر کرده باشند. پس با انسامبل کردن چندین مدل (ترجیحا با میانگین‌گیری) که به لوکال مینیمم‌ها رسیده‌اند، پیشبینی ما بهبود می‌یابد. برای اینکه ارور این مدل‌ها، مستقل از هم باشد، میتوانیم از مدلهایی با ساختار متفاوت استفاده کنیم. مثلا تعداد لایه یا نورون یا لرنینگ ریت یا رگولارایزیشن آنها متفاوت باشد.
گاهی، یک مدل داریم که اموزش آن هفته‌ها طول کشیده. در این موارد نیز بهترین حالات مدل را در طی فرایند اموزش بصورت دوره‌ای، سیو میکنند و سپس یک انسامبل روی این مدل‌های بدست آمده اجرا میکنند که به آن Snapshot Ensembling میگویند. در واقع

انسامبل روی مدل درون لوکال مینیمم‌ها گرفته میشود.
در مورد روش‌های ترکیب مدل‌های ضعیف، روش معمول میانگینگیری است. یا میتوانیم به هر خروجی یک وزن نسبت دهیم و میانگین وزندار بگیریم. که خود وزن‌ها learnable اند. یعنی برای ترکیب، یک مدل خطی داشته باشیم. این مدل میتواند غیرخطی هم باشد که همان استکینگ است. یک روش دیگر هم این است ه اگر ساختار مدلهای ضعیف یکسان بود، وزن‌های داخل شبکه‌ها را ترکیب کنیم.

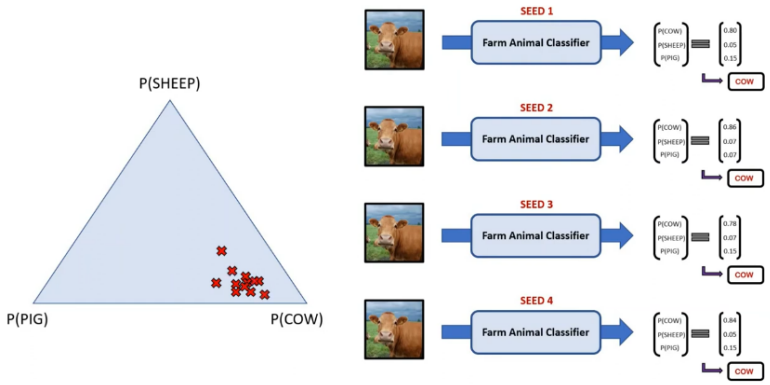
۷

عدم اطمینان در شبکه یعنی اینکه چقدر میتوانیم به پاسخ شبکه اطمینان کنیم و چقدر نه. اگر از انسان‌ها سوالی بپرسیم و او جواب را نداند، میگوید "نمیدانم، ولی حدس میزنم پاسخ X یا Y باشد." اگر مدل ما هم بتواند یک مجموعه از جواب‌های محتمل را پیشنهاد دهد، بسیار قابل اعتمادتر است. و اگر بتواند مثلا بگوید به احتمال ۹۰ درصد پاسخ سوالی این است، بسیار در فهم ما مفید خواهد بود. مثلا فرض کنید یک مدل بسازیم که با گرفتن یک عکس رادیوگرافی، صرفا نام یک بیماری خروجی دهد و یک مدل دیگر هم بسازیم که نام ۲ بیماری و درصد احتمال وجود این بیماریها را بدهد. قطعا مدل دوم بسیار به پزشک کمک بیشتری میکند. در این مسائل کلسیفیکیشن، یک معیار برای اندازه‌گیزی میزان عدم اطمینان، انتروپی است. هرچه انتروپی بالاتر، عدم اطمینان بیشتر.
عدم اطمینان در کل، مجموع عدم اطمینان مدل و عدم اطمینان دیتاست. عدم اطمینان در دیتا، یعنی خود دیتای ورودی، مشخص نیست متعلق به کدام کلاس است. مثلا یک عکس مبهم از حیوانی که خود ما هم بعنوان انسان نمیتوانیم بفهمیم سگ است یا گربه. یا مثلا یک نویز در دیتای خطی. به این نوع از عدم اطمینان aleatoric uncertainty گفته میشود.



شکل ۶: aleotoric uncertainty: اینجا وقتی هوا سرد باشد، با اطمینان بالا میتوان گفت فروش بستنی کم میشود اما وقتی هوا گرم باشد، با اطمینان کم میتوان گفت که فروش چقدر بالا میرود چون نقاط پراکنده‌ترند. همچنین برای دمایی بزرگتر از آنچه در نمودار داریم، هیچ دیتایی نیست (out of distribution) و اگر درآنجا پیشبینیی انجام دهیم، اطمینان کمی داریم.

در مقابل، وقتی عدم اطمینان مربوط به مدل آموزش داده شده باشد، epistemic uncertainty داریم که با دیتای بیشتر، هایپرپارامترهای بهتر و یادگیری بهتر مدل، قابل رفع است.

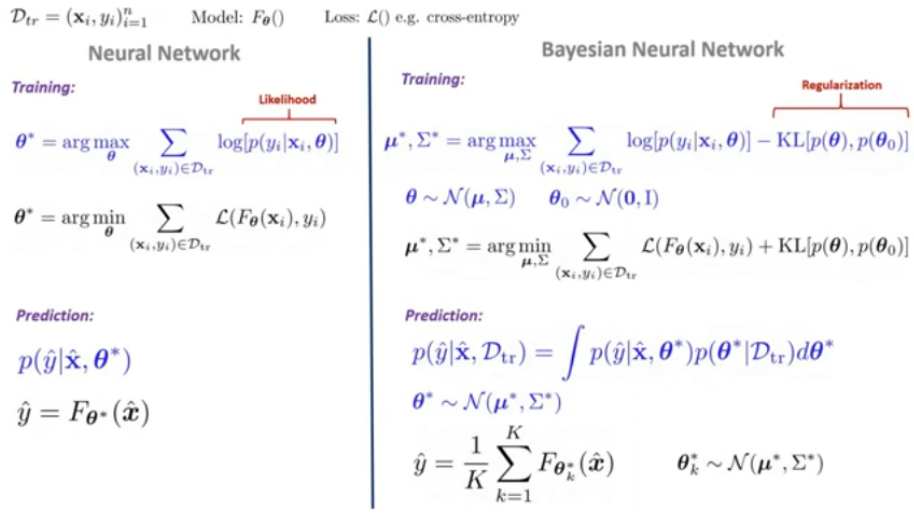


شکل ۷: model uncertainty

در شکل ۷، ما هر چند بار با وزن‌های اولیه مختلف، مدلمان را آموزش دهیم، به چند مدل نهایی مختلف میرسیم که هر کدام برای یک ورودی (عکس گاو) یکسان، جوابهای متفاوتی میدهند. در واقع خود این مدل‌ها، سمپل‌هایی از یک توزیع احتمالاتی هستند و آن توزیع احتمالاتی، جوابهای مسئله ما را در بر دارد. واریانس این توزیع هم نشان میدهد چقد مدل‌های ما با هم موافقت میکنند.

برای اندازه‌گیری عدم اطمینان اپیستمیک، میتوانیم از دراپ‌اوت مونته‌کارلو یا مدل بیزی استفاده کنیم.

در نورال نتورک ساده، ما به دنبال یافتن وزن‌ها هستیم. اما در یک نورال نتورک بیزی، ما دنبال یافتن یک توزیع احتمالاتی به ازای هر وزن هستیم که آن وزن از این توزیع نمونه‌گیری بشود. مقلا این توزیع‌ها را گاوسی در نظر میگیریم و به دنبال یک واریانس و یک میانگین برای هر وزن هستیم. در واقع ما با یافتن توزیع این وزن‌ها، یک انسامبل از بی‌نهایت شبکه را خواهیم داشت.



شکل ۸: تفاوت یک مدل بیزی با مدل عادی

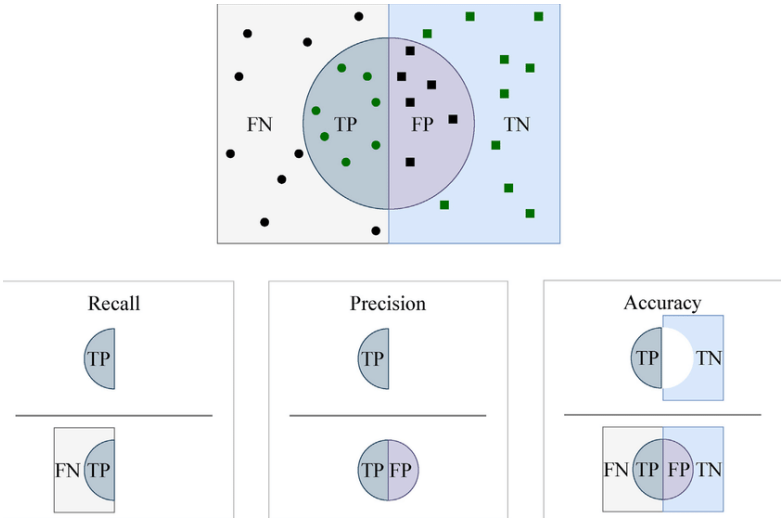
هرباری که ورودی به یک شبکه بیزی داده شود، وزن‌ها از توزیع خودشان نمونه‌گیری میشوند. این یعنی وزن‌ها هربار متفاوتند و به ازای ورودی یکسان، هربار خروجی فرق میکند. برای یافتن مقدار epistemic/model uncertainty میتوانیم چندبار نمونه‌گیری کنیم و ببنیم در پیشبینی‌ها چقدر واریانس داشته‌ایم. اما استفاده از شبکه بیزی بسیار هزینه‌بر است هم در آموزش و هم در تست.میتوان بجای آن، از مدلی با MC dropout استفاده کرد.

در یک مدل با دراپ‌اوت مونته کارلو، هر بار که تست را به آن میدهیم یکی از مدلهای درون توزیع احتمالاتی به ما جواب میدهد. و در طول آموزش، ما در واقع توزیع احتمالاتی هر وزن را لرن کرده‌ایم، نه مقدار دقیق هر وزن برای دادن جواب.

مدلی با MC dropout در واقع یک انسامبل از شبکه‌هایی با وزن‌هایی یکسان است. واریانس پیشبینی‌های این شبکه به ازای یک ورودی یکسان، همان epistemic/model uncertainty است.

۸

accuracy برابر تعداد پیشبینی‌های درست مدل (true positives + true negatives)تقسیم بر تعداد کل پیشبینی‌های انجام‌شده است. حالا اگر در دیتایی که به مدل داده‌ایم، مثلا ۹۵ درصدشان از کلاس ۱ و ۵ درصد کلاس ۲ باشند، حتی اگر مدل ما همیشه ۱ را خروجی بدهد accuracy ۹۵ درصد میشود. وقتی کلاسها بالانس نباشند یا نویز داشته باشیم باید معیارهای دیگری هم داشته باشیم تا مدل را ارزیابی کنیم. از جمله این معیارها، precision و recall هستند. هر دوی اینها برای اندازه‌گیری عملکرد مدل روی کلاسی که در اقلیت است مناسبند.



شکل ۹: recall, precision and accuracy

precision تعداد داده‌هایی که به درستی بعنوان "یک" کلسیفای شدند تقسیم بر تعداد کل داده‌هایی که بعنوان "یک" کلسیفای شدند.
recall تعداد داده‌هایی که به درستی بعنوان "یک" کلسیفای شدند تقسیم بر تعداد کل داده‌هایی که در حقیقت "یک" هستند.

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

یکی دیگر از مقادیری است که میتواند در ارزیابی مدل مفید باشد. همچنین نمودار ROC که true positive rate=$\frac{TP}{TP+FN}$ محور عمودی و false positive rate = $\frac{FP}{FP+TN}$ محور افقی آن است و این دو متغیر را در classification threshold های مختلف رسم میکند. باید مساحت زیر نمودار بالاترین حد خودش باشد و این مساحت یعنی AUC خودش یک معیار ارزیابی مدل است که فارغ از بالانس بودن دیتا کار میکند.

۹

هر بار که ما در فرایند SGD گرادیان را محاسبه میکنیم، این گرادیان را روی بخشی از دیتا محاسبه کرده‌ایم پس همیشه یک خطایی داریم و همین باعث ذات تصادفی بودن و نویزی و غیرهمواری loss در الگوریتم SGD میشود، مگر اینکه به مرور زمان، Learning Rate را کوچک و کوچکتر کنیم تا نگذاریم دیتاهای بعدی (وقتی هنوز loss صفر نشده اما بسیار

کوچک شده) دوباره ما را از اپتیمم منحرف و دورتر کنند. در ضمن، هرچقدر batch ها بزرگتر باشند این خطا و این تصادفی بودن کمتر میشود و به گرادیان واقعی نزدیکتر میشویم.

The Classical Convergence Theorem

$$\Phi \mathrel{-}= \eta_t \nabla_\Phi \operatorname{loss}(\Phi, x_t, y_t)$$

For "sufficiently smooth" non-negative loss with

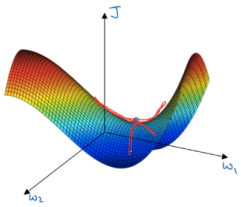
$$\eta_t \geq 0 \qquad \lim_{t\to\infty} \eta_t = 0 \qquad \sum_t \eta_t = \infty \qquad \sum_t \eta_t^2 < \infty$$

we have that the training loss $E_{(x,y)\sim\text{Train}} \operatorname{loss}(\Phi, x, t)$ converges to a limit and any limit point of the sequence Φ_t is a stationary point in the sense that $\lim_t \nabla_\Phi E_{(x,y)\sim\text{Train}} \operatorname{loss}(\Phi, x, t) = 0$.

Rigor Police: One can construct cases where Φ diverges to infinity, converges to a saddle point, or even converges to a limit cycle.

شکل ۱۰: این قضیه، شروط همگرایی loss در الگوریتم SGD بیان میکند. و میگوید که اگر وزنها همگرا شوند، به جایی همگرا میشوند که loss صفر شود. اما این مکان میتواند مینیمم نباشد و مثلا یک saddle point باشد.

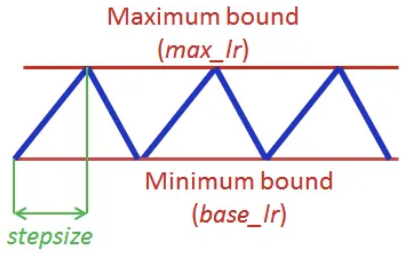
طبق این قضیه، میتوانیم بی‌نهایت بار با لرنینگ ریت بسیار کوچکی حرکت کنیم تا گرادیان نهایتا به گرادیان واقعی نزدیک شود.
در واقع، همگرایی loss به صفر در الگوریتم SGD به چند فاکتور وابسته است:
■learning rate: اگر آن را خیلی بزرگ انتخاب کنیم یعنی برای هر بچ، اهمیت بسیار بالایی قائل شده‌ایم و این باعث میشود نوسان بالا برود و به راحتی از روی اپتیمم بپریم. البته اگر بسیار کوچک باشد، فقط همگرایی را کند میکند.
■ اینکه خود تابع loss باید smooth و convex باشد. روی توابعی که non convex باشند، هیچ تضمینی برای رسیدن به مینیمم گلوبال نیست و بسیار محتمل است در saddle point گیر بیفتیم. در حالی که در توابع convex این تضمین وجود دارد.
■سایز بچ‌ها هرچقدر بزرگتر باشد، گرادیان به گرادیان واقعی نزدیکتر است اما همگرایی ممکن است کند شود.
برای آدام، همگرایی به عوامل زیر بستگی دارد:
■learning rate و پارامتر momentum
■وجود فاکتورهایی مثل saddle point درون تابع.
درواقع آدام نسخه‌ای بهتر از SGD است که همچنان برای توابع هزینه‌ی محدب خوب کار میکند و همچنین در توابع غیر محدب هم میتواند همگرا شود چون که لرنینگ ریت آداپتیو است.



شکل ۱۱: saddle point

برای فرار از نقاطی که گرادیان نزدیک صفر دارند ولی مینیمم نیستند(saddle points) یا لوکال مینیمم‌ها یا جایی که گرادیان برای مدت زیادی نزدیک به صفر مانده، روش‌های مختلفی داریم:

- اینکه لرنینگ ریت با گذر زمان کوچکتر شود. در ابتدا با مقداری بزرگ شروع میکنیم تا راحت بتوانیم explore کنیم سپس با لرنینگ ریت کوچکی باید دنبال exploit کردن باشیم. مثلا با کمک Step decay.
- لرنینگ ریت سیکلی. این بسیار در حین مواجهه با saddle point ها مفید است و کمک میکند به سرعت از آنها فرار کنیم.



شکل ۱۲: cyclic learning rate

- استفاده از آدام یا یک لرنینگ ریت آداپتیو.
- استفاده از تکنیک‌های خوب برای وزن‌دهی اولیه مثل xavier هم موثر است.
- Perturbed Gradient Descent یا همان گرادیان تصادفی نویزدار. به این صورت است که به گرادیان‌ها یک نویز اضافه میکنیم تا به آن کمک کنیم از سدل پوینت‌ها راحتتر فرار کند و همگرایی سریعتر شود.
- الگوریتم‌های درجه دوم بهینه‌سازی مثل نیوتن نیز میتوانند کمک کنند.

روش نیوتن، یک روش iiterative است که هم برای ریشه‌یابی و هم بهینه‌سازی کاربرد دارد. در این روش ابتدا با یک حدس رندوم اولیه شروع میکنیم. سپس با یک بردار، آن را انتقال میدهیم و در این نقطه جدید، مقدار تابع را محاسبه میکنیم که امیدواریم کمتر شده باشد. این چرخه چندین و چند بار تکرار میشود. برای یافتن برداری که با آن در جهت بهینه‌سازی حرکت کنیم هم میتوانیم از اطلاعات گرادیان تابع در نقطه اول و ماتریس هسین آن استفاده کنیم. ماتریس هسین، curvature را نشان میدهد. هر نقطه بدین صورت بدست می‌آید:

$$x_{k+1} = x_k - 1\frac{1}{f''(x_k)} \times f'(x_k)$$

در کل روش‌های درجه دوم، از ماتریس هسین تابع هزینه استفاده میکنند یعنی بجای اینکه مثل گرادیان کاهشی، یک درجه مشتق بگیرند، دوبار مشتق میگیرند. روش نیوتن سریعتر همگرا میشود نسبت به گرادیان کاهشی. تعداد ارقام صحیح و دقیق، حدودا با هر iteration دوبرابر میشود. نکته مثبت دیگر این است که اینجا یک learning rate مثل آلفا نداریم. اما یک عیب آن این است که بسته به نقطه اولیه، ممکن است بیشتر طول بکشد تا به جواب برسیم یا جواب موردنظر را پیدا نکنیم. همچنین از لحاظ محاسباتی هزینه‌بر است چون ماتریس هسین باید محاسبه شود. روش‌های quasi-newton از گرادیان برای محاسبه هسین استفاده میکنند که باعث میشود در عین همگرایی سریع، محاسبات اسان‌تری هم داشته باشند. در واقع در این روشها، ماتریس هسین را در ابتدا بطور تخمینی ماتریس همانی میگیریم و در طی بهینه‌سازی این ماتریس را هم آپدیت میکنیم.