**فصل 21 - شبکه‌های عصبی**

**21. 0 مقدمه**

در مرکز شبکه‌های عصبی واحدی به نام گره یا بخش مرکزی نورون قرار دارد. گره یک یا چند ورودی دریافت می‌کند، هر ورودی را به یک پارامتر (که به وزن نیز معروف است) دیگر ضرب می‌کند، مقادیر ورودی وزن‌دار را به همراه مقدار تعیین‌شده‌ای به نام بایاس (معمولاً 1) جمع می‌کند، سپس این مقدار به تابع فعال‌ساز وارد می‌شود. سپس این خروجی به سمت نورون‌های دیگر در اعماق شبکه عصبی هستند فرستاده می‌شود (اگر وجود داشته باشند).

شبکه‌های عصبی را می‌توان به‌عنوان مجموعه‌ای از لایه‌های متصل تجسم کرد که شبکه‌ای را تشکیل می‌دهند که مقادیر ویژگی‌های یک مشاهده را در یک انتها و مقدار هدف (به عنوان مثال، کلاس مشاهده) را در انتهای دیگر به هم متصل می‌کند. شبکه‌های عصبی پیشخور - که پرسپترون چندلایه[[1]](#footnote-1) نیز نامیده می‌شوند، ساده‌ترین شبکه‌های عصبی مصنوعی هستند که در هر محیط واقعی مورد استفاده قرار می‌گیرند. نام feedforward از این واقعیت ناشی می شود که مقادیر ویژگی مشاهده از طریق شبکه «به جلو» تغذیه می‌شود، با هر لایه به طور متوالی مقادیر ویژگی را با این هدف که خروجی مشابه (یا نزدیک به) مقدار هدف باشد، تغییر می دهد.

به طور خاص، شبکه‌های عصبی پیشخور شامل سه نوع لایه از واحدها هستند. در ابتدای این شبکه عصبی، لایه ورودی قرار دارد که هر واحد شامل مقدار مشاهده برای یک ویژگی است. به عنوان مثال، اگر مشاهده دارای 100 ویژگی باشد، لایه ورودی 100 گره دارد. در انتهای شبکه عصبی لایه خروجی قرار دارد که خروجی لایه‌های پنهان را به مقادیری مفید برای وظیفه مورد نظر تبدیل می‌کند. به عنوان مثال، اگر هدف ما طبقه‌بندی دودویی باشد، می‌توانیم از لایه خروجی با یک واحد استفاده کنیم که از تابع سیگموئید برای مقیاس دادن خروجی خود به مقداری بین 0 و 1 استفاده کند، که احتمال کلاس پیش‌بینی‌شده را نمایان می‌سازد. بین لایه ورودی و خروجی، لایه‌های مخفی به نام hidden layers (که به هیچ وجه مخفی نیستند) قرار دارند. این لایه‌های مخفی به ترتیب مقادیر ویژگی را از لایه ورودی به چیزی تغییر می‌دهند که پس از پردازش توسط لایه خروجی، به شکل کلاس هدف شبیه شود.

شبکه‌های عصبی‌ای که دارای لایه‌های مخفی زیادی (مانند 10، 100، 1000) هستند به عنوان شبکه‌های «عمیق» محسوب می‌شوند و کاربرد آنها به عنوان یادگیری عمیق شناخته می‌شود.

شبکه‌های عصبی به طور معمول با تمام پارامترها با مقادیر تصادفی کوچک از توزیع گوسی یا توزیع یکنواخت نرمال مقداردهی اولیه می‌شوند. پس از اینکه مشاهده‌ای (یا بیشتر، معمولاً تعداد تعیین‌شده‌ای از مشاهدات به نام دسته) از طریق شبکه انتقال داده می‌شود، مقدار خروجی با مقدار واقعیِ مشاهده شده با استفاده از تابع خطا مقایسه می‌شود. این به عنوان پیش‌رَوی انجام می‌شود. سپس یک الگوریتم به طور پَسرو از طریق شبکه حرکت می‌کند و می‌شناسد که هر پارامتر چقدر به اشتباه بین مقدار پیش‌بینی شده و مقدار واقعی کمک کرده است که به عمل فرآیند برگشت مشهور است. در هر پارامتر، الگوریتم بهینه‌سازی تعیین می‌کند که هر وزن باید چقدر تغییر یابد تا خروجی بهبود یابد. شبکه‌های عصبی با تکرار این فرآیند از پیش‌رَویی و برگشت برای هر مشاهده در داده‌های آموزش چندین بار (هر بار که همه مشاهدات از طریق شبکه ارسال شده باشند، به عنوان یک دوره شناخته می‌شود و آموزش معمولاً از چندین دوره تشکیل شده است) مقادیر پارامترها را به‌روز می‌کنند. در این فصل، ما از کتابخانه محبوب پایتون به نام Keras برای ساخت، آموزش و ارزیابی انواع مختلف شبکه‌های عصبی استفاده خواهیم کرد. Keras یک کتابخانه سطح بالا است و از کتابخانه‌های دیگری مانند TensorFlow و Theano به عنوان موتور خود استفاده می‌کند. برای ما مزیت Keras این است که می‌توانیم بر روی طراحی و آموزش شبکه تمرکز کنیم و جزئیات عملیات تانسورها را به کتابخانه‌های دیگر واگذار کنیم. شبکه‌های عصبی ساخته‌شده با استفاده از کد Keras می‌توانند با استفاده از همه‌ی پردازنده‌ها (یعنی حتی در لپ‌تاپ شما) و همچنین با پردازنده‌های گرافیکی (یعنی در یک کامپیوتر یادگیری عمیق ویژه) آموزش داده شوند. در دنیای واقعی با داده‌های واقعی، به شدت توصیه می‌شود که شبکه‌های عصبی با استفاده از پردازنده‌های گرافیکی آموزش داده شوند؛ با این حال، به منظور یادگیری، همه شبکه‌های عصبی در این کتاب به اندازه کافی کوچک و ساده هستند تا در چند دقیقه در لپ‌تاپ شما آموزش داده شوند. فقط به خاطر داشته باشید که زمانی که شبکه‌های بزرگتر و داده‌های آموزش بیشتری داشته باشیم، آموزش با استفاده از پردازنده‌های مرکزی به طرز قابل توجهی کندتر از آموزش با استفاده از پردازنده‌های گرافیکی است.

**1.21 استفاده از Autograd با PyTorch**

**مسئله**

می‌خواهید از ویژگی‌های خودکار PyTorch برای محاسبه و ذخیره گرادیان‌ها پس از انتشار به جلو و انتشار به عقب استفاده کنید.

**راه‌حل**

Tensorها را با گزینه requires\_grad که روی True تنظیم شده است ایجاد کنید:



**بحث**

Autograd یکی از ویژگی‌های اصلی PyTorch و عامل مهمی ‌در محبوبیت آن به عنوان یک کتابخانه یادگیری عمیق است. توانایی محاسبه، ذخیره و تجسم آسان گرادیان ها، PyTorch را برای محققان و علاقه مندان به ساخت شبکه‌های عصبی از ابتدا، بسیار شهودی می‌کند.

PyTorch از یک گراف غیر چرخه‌ای جهت دار (DAG) برای نگه داشتن رکوردی از تمام داده‌ها و عملیات محاسباتی انجام شده بر روی آن داده استفاده می‌کند. این امر فوق‌العاده مفیدی است، اما به این معنی است که ما باید مراقب باشیم که چه عملیاتی را روی داده‌های PyTorch خود اعمال می‌کنیم که به گرادیان نیاز دارند. هنگام کار با autograd، نمی‌توانیم به راحتی Tensorهای خود را بدون «شکستن نمودار»[[2]](#footnote-2) به آرایه‌های NumPy تبدیل کنیم، عبارتی که برای توصیف عملیات‌هایی که از autograd پشتیبانی نمی‌کنند استفاده می‌شود:



برای تبدیل این Tensor به یک آرایه NumPy، باید متد ()detach را روی آن فراخوانی کنیم، که نمودار را شکسته و در نتیجه توانایی ما برای محاسبه خودکار گرادیان‌ها را می‌شکند. در حالی که این کار قطعا می‌تواند مفید باشد، ارزش دانستن این را دارد که جدا کردن Tensor، از محاسبه خودکار گرادیان PyTorch جلوگیری می‌کند.

**21.2 پیش پردازش داده برای شبکه‌های عصبی**

**مسئله**

شما می‌خواهید داده‌ها را برای استفاده در یک شبکه عصبی، پردازش کنید.

**راه‌حل**

هر ویژگی را با استفاده از استاندارد Scaler در scikit-learn استاندارد کنید:



**بحث**

اگرچه این دستورالعمل بسیار شبیه به دستورالعمل 4.3 است، ارزش تکرار آن به دلیل اهمیت آن برای شبکه‌های عصبی حائز اهمیت است. به طور معمول، پارامترهای یک شبکه عصبی به عنوان اعداد تصادفی کوچک، مقداردهی اولیه می‌شوند. شبکه‌های عصبی معمولاً زمانی که مقادیر ویژگی‌ها به طور قابل توجهی بزرگتر از مقادیر پارامترها هستند به شکل ناپسندی عمل می‌کنند. علاوه بر این، از آنجا که مقادیر ویژگی‌های یک مشاهده به عنوان آنها از طریق واحدهای تکی ترکیب می‌شوند، اهمیت دارد که تمام ویژگی‌ها از همان مقیاس استفاده کنند.

با توجه به این دلایل، بهترین شیوه (اگرچه همیشه ضروری نیست؛ به عنوان مثال، زمانی که تمام ویژگی‌ها دارای ویژگی‌های دودویی هستند) توصیه می‌شود که هر ویژگی را به نحوی استاندارد کنیم که مقادیر ویژگی‌ها میانگین 0 و انحراف معیار 1 داشته باشند. این کار به راحتی با استفاده از ابزار StandardScaler در scikit-learn قابل انجام است.

تأثیر استانداردسازی را می‌توانید با بررسی میانگین و انحراف معیار ویژگی‌های اولیه ما مشاهده کنید:



**3.21 طراحی شبکه عصبی**

**مسئله**

شما می­خواهید یک شبکه عصبی طراحی کنید.

**راه‌حل**

از مدل Sequential در Keras استفاده کنید:



شبکه‌های عصبی از لایه‌هایی متشکل از یونیت‌ها یا همان گره‌ها تشکیل شده‌اند. با این حال، تنوع شگفت‌انگیزی در انواع لایه‌ها و چگونگی ترکیب آن‌ها برای ایجاد معماری شبکه وجود دارد. در حال حاضر، با وجود الگوهای معماری معمولی (که در این فصل به آن‌ها خواهیم پرداخت)، حقیقت این است که انتخاب معماری مناسب در اغلب موارد یک هنر است و موضوع تحقیقات فراوانی است. برای ساخت یک شبکه عصبی پیش‌خور در Keras، باید در مورد معماری شبکه و فرآیند آموزش، تعدادی انتخاب انجام دهیم. به خاطر داشته باشید که هر واحد در لایه‌های مخفی:

تعدادی ورودی دریافت می‌کند.

هر ورودی را با مقدار یک پارامتر وزن‌دهی می‌کند.

تمام ورودی‌های وزن‌دار را به همراه تعدادی تغییر بایاس (معمولاً 1) جمع می‌کند.

بیشتر اوقات تابعی (که تابع فعال‌ساز نامیده می‌شود) را روی ورودی اعمال می‌کند.

خروجی را به یونیت‌ها یا همان گره‌ها در لایه‌ی بعدی ارسال می‌کند. ابتدا، برای هر لایه در لایه‌های مخفی و خروجی، باید تعداد واحدهایی که در لایه قرار می‌دهیم و تابع فعال‌ساز را تعیین کنیم. به طور کلی، هر چه تعداد واحدها در یک لایه بیشتر باشد، شبکه ما قادر به یادگیری الگوهای پیچیده‌تر خواهد بود. با این حال، تعداد بیشتر یونیت‌ها ممکن است باعث بیش‌برازش شبکه به داده‌های آموزشی شده و به نحوی مخرب، بر عملکرد داده‌های آزمایشی تأثیر بگذارد. برای لایه‌های مخفی، تابع فعال‌ساز محبوب واحد خطی تصحیح شده (ReLU) است:

که z مجموع ورودی‌های وزن‌دار و انحراف است. همان‌طور که می‌بینیم، اگر z از 0 بزرگتر باشد، تابع فعال‌ساز مقدار z را برمی‌گرداند؛ در غیر اینصورت، تابع مقدار 0 را باز می‌گرداند. این تابع فعال‌ساز ساده دارای تعدادی خاصیت مطلوب است (که **بحث** درباره‌ی آن خارج از دامنه‌ی این کتاب است) و این امر باعث شده ‌است که انتخاب محبوبی در شبکه‌های عصبی باشد. با این حال، باید آگاه باشیم که تعداد زیادی تابع فعال‌ساز وجود دارد. ثانیاً، باید تعداد لایه‌های مخفی را در شبکه تعیین کنیم. لایه‌های بیشتر، به شبکه امکان یادگیری روابط پیچیده‌تر را می‌دهد، اما با هزینه‌ی محاسباتی بیشتری همراه است. سوماً، ما باید ساختار تابع فعال‌ساز (در صورت وجود) لایه خروجی را تعیین کنیم. ماهیت تابع خروجی اغلب توسط هدف شبکه تعیین می‌شود. در ادامه، برخی از الگوهای رایج لایه خروجی آمده است:

طبقه‌بند دوتایی (Binary classification)

یک واحد با تابع فعال‌ساز سیگموئید.

طبقه‌بند چنددرجه‌ای (Multiclass classification)

k واحد (که k تعداد کلاس‌های هدف است) و تابع فعال‌ساز softmax.

رگرسیون (Regression)

یک واحد بدون تابع فعال‌ساز.

چهارم، ما باید یک تابع خطا (تابعی که میزان تطابق مقدار پیش‌بینی شده با مقدار واقعی را اندازه‌گیری می‌کند) تعریف کنیم؛ این تابع معمولاً با توجه به نوع مسئله تعیین می‌شود:

طبقه‌بند دوتایی

آنتروپی متقابل دوتایی. (Binary cross-entropy)

طبقه‌بند چنددرجه‌ای

آنتروپی رسته‌ای. (Categorical cross-entropy)

رگرسیون

خطای میانگین مربعات. (Mean square error)

پنجم، ما باید یک بهینه‌ساز تعریف کنیم، که به طور ساده می‌توان آن را به عنوان استراتژی‌مان برای پیمایش تابع خطا برای یافتن مقادیر پارامترهایی که کمترین خطا را ایجاد می‌کنند، تصور کرد. انتخاب‌های رایج برای بهینه‌سازها شامل گرادیان کاهشی تصادفی[[3]](#footnote-3)، گرادیان کاهشی تصادفی با تکانه[[4]](#footnote-4)، پخش جذر میانگین مربعات[[5]](#footnote-5) و برآورد لحظه‌ای تطبیقی[[6]](#footnote-6) هستند (اطلاعات بیشتر در همچنین ببینید).

ششم، ما می‌توانیم یک یا چند معیار برای ارزیابی عملکرد انتخاب کنیم، مانند دقت.

در این مثال، ما از یک namespace به نام torch.nn.Module برای ایجاد یک شبکه عصبی ساده و متوالی استفاده می کنیم که می تواند طبقه‌بندی‌های باینری انجام دهد. رویکرد استاندارد PyTorch برای این کار، ایجاد یک کلاس فرزند است که کلاس torch.nn.Module را به ارث می‌برد. نمونه‌سازی یک معماری شبکه در روش \_\_init\_\_ و تعریف عملیات ریاضی که می‌خواهیم بر روی هر عبور به جلو در روش فوروارد در کلاس انجام دهیم. راه‌های زیادی برای تعریف شبکه‌ها در PyTorch وجود دارد، و اگرچه در این مورد ما از روش‌های تابعی برای توابع فعال‌سازی خود (مانند nn.functional.relu) استفاده می‌کنیم، می‌توانیم این توابع فعال‌سازی را نیز به صورت یک لایه تعریف کنیم. اگر بخواهیم همه چیز را در شبکه به صورت یک لایه بنویسیم، می‌توانیم از کلاس Sequential استفاده کنیم:



در هر دو مورد، خود شبکه، یک شبکه عصبی دو لایه است (هنگام شمارش لایه‌ها، لایه ورودی را در نظر نمی‌گیریم زیرا هیچ پارامتری برای یادگیری ندارد) که با استفاده از مدل ترتیبی PyTorch تعریف شده است. هر لایه "متراکم" است (همچنین "کاملا متصل" نیز نامیده می شود)، به این معنی که تمام واحدهای لایه قبلی به تمام واحدهای لایه بعدی متصل هستند. در اولین لایه پنهان، out\_features=16 را تنظیم کردیم، به این معنی که لایه شامل 16 واحد است. این واحدها دارای توابع فعالسازی ReLU هستند که در روش فوروارد در کلاس ما تعریف شده است: x = nn.functional.relu(self.fc1(x)). اولین لایه شبکه‌ی ما دارای اندازه‌ی (10، 16) است، که به لایه اول می گوید که انتظار داشته باشد هر مشاهده از داده های ورودی ما دارای 10 مقدار ویژگی باشد. این شبکه برای طبقه‌بندی باینری طراحی شده است، بنابراین لایه خروجی تنها دارای یک واحد با تابع فعال‌سازی سیگموئید است که خروجی را بین 0 و 1 محدود می‌کند (که احتمال مشاهده کلاس 1 را نشان می‌دهد).

**همچنین ببینید:**

* [آموزش PyTorch: ساخت شبکه عصبی](https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html)
* [توابع خطا برای طبقه‌بندی، ویکی‌پدیا](https://en.wikipedia.org/wiki/Loss_functions_for_classification)
* [در مورد توابع خطا برای شبکه‌های عصبی عمیق در طبقه‌بندی، کاتارژینا یانوخا، وویچک ماریان چارنتسکی](https://arxiv.org/abs/1702.05659)

**4.21 آموزش یک طبقه‌بند دوتایی**

**مسئله**

شما می‌خواهید یک شبکه عصبی طبقه‌بند دوتایی آموزش دهید.

**راه‌حل**

از Keras برای ساخت یک شبکه عصبی روبه جلو(Feed Forward) استفاده کرده و آن را با استفاده از متد fit آموزش دهید.



**بحث**

در دستور 21.3، نحوه‌ی ساخت یک شبکه‌ی عصبی با استفاده از مدل ترتیبی PyTorch را مورد بحث قرار دادیم. در این دستور العمل، ما آن شبکه عصبی را با استفاده از 10 ویژگی و 1000 مشاهده‌ی طبقه بندی جعلی ایجاد شده از تابع make\_classification در scikit-learn آموزش می‌دهیم.

شبکه عصبی مورد استفاده‌ی ما همان شبکه ای است که در دستور العمل 21.3 وجود دارد (برای توضیح دقیق به آن دستور العمل مراجعه کنید). تفاوت اینجاست که ما فقط شبکه‌ی عصبی را ایجاد کردیم. ما آن را آموزش ندادیم

در انتها از torch.no\_grad() برای ارزیابی شبکه استفاده می کنیم. این تابع می گوید که ما نباید گرادیان‌ها را برای عملیات تانسور انجام شده در این بخش از کد محاسبه کنیم. از آنجایی که ما از شیب ها فقط در طول فرآیند آموزش مدل استفاده می کنیم، نمی خواهیم شیب های جدیدی را برای عملیاتی که خارج از آن رخ می دهد (مانند پیش بینی یا ارزیابی) ذخیره کنیم.

متغیر epochs تعیین می کند که از چند دوره برای آموزش داده ها استفاده شود. batch\_size تعداد مشاهدات را برای انتشار در شبکه قبل از به روز رسانی پارامترها تنظیم می کند.

سپس تعداد دوره‌ها را تکرار می‌کنیم، با استفاده از روش رو به جلو از شبکه عبور می‌کنیم و سپس برای به‌روزرسانی گرادیان‌ها، پاس‌های رو به عقب را انجام می‌دهیم. نتیجه یک مدل آموزش دیده است.

**5.21 آموزش یک طبقه‌بندی کننده‌ی چند کلاسه**

**مسئله**

شما می­خواهید یک شبکه عصبی طبقه‌بند چند کلاسه را آموزش دهید.

**راه‌حل**

از PyTorch برای ساخت یک شبکه عصبی پیشخور با یک لایه خروجی با توابع فعال سازی softmax استفاده کنید:



**بحث**  
در این راه حل، ما یک شبکه عصبی مشابه با طبقه بندی کننده باینری از آخرین دستور، اما با تغییرات قابل توجه ایجاد کردیم. در داده های طبقه بندی‌ای که تولید کردیم، N\_CLASSES=3 را تنظیم کردیم. برای رسیدن به طبقه‌بندی چند کلاسه، ما همچنین از nn.CrossEntropyLoss() استفاده می‌کنیم که انتظار دارد هدف، یک‌بار کدگذاری شود. برای انجام این کار، از تابع torch.nn.functional.one\_hot استفاده می کنیم و در نهایت منتهی می‌شویم به یک آرایه ی رمزگذاری شده که در موقعیت 1 قرار دارد که یک مشاهده مشخص نشان را می‌دهد:



از آنجایی که این یک مشکل طبقه بندی چند کلاسه است، ما از یک لایه خروجی با اندازه 3 (یکی برای هر کلاس) استفاده کردیم که حاوی تابع فعال سازی softmax است. تابع فعال‌سازی softmax آرایه‌ای از 3 مقدار را برمی‌گرداند که جمع آن 1 است. این 3 مقدار احتمال یک مشاهده را برای عضویت در هر یک از 3 کلاس نشان می‌دهند.

همانطور که در این دستور ذکر شد، ما از یک تابع ضرر مناسب برای طبقه‌بندی چند کلاسه استفاده کردیم، تابع ضرر آنتروپی طبقه‌ای: nn.CrossEntropyLoss().

**6.21 آموزش یک مدل رگرسیون**

**مسئله**  
شما می­خواهید یک شبکه عصبی را برای رگرسیون آموزش دهید.

**راه‌حل**

از PyTorch برای ساخت یک شبکه عصبی پیشخور با یک واحد خروجی که هیچ تابع فعال سازی ندارد استفاده کنید:



**بحث**  
ایجاد یک شبکه عصبی برای پیش‌بینی مقادیر پیوسته به جای احتمالات کلاس کاملاً امکان‌پذیر است. در مورد طبقه‌بندی‌کننده‌ی دودویی (دستور العمل 21.3) ما از یک لایه خروجی با یک واحد و یک تابع فعال‌ساز سیگموئید استفاده کردیم تا احتمال اینکه مشاهده‌ای کلاس 1 باشد را ایجاد کنیم. نکته مهم این است که تابع فعال‌ساز سیگموئید مقدار خروجی را بین 0 و 1 محدود می‌کند. اگر آن محدودیت را با نداشتن تابع فعال‌سازی حذف کنیم، اجازه می­دهیم خروجی یک مقدار پیوسته باشد.

علاوه بر این، از آنجایی که ما در حال آموزش رگرسیون هستیم، باید از یک تابع ضرر و معیار ارزیابی مناسب استفاده کنیم. در مورد ما میانگین مربعات خطا:

که در آن n تعداد مشاهدات است. مقدار واقعی هدفی است که می‌خواهیم پیش‌بینی کنیم، y، برای مشاهده iام و مقدار پیش بینی شده مدل برای است.

در نهایت، چون ما از داده­های شبیه‌سازی شده با استفاده از make\_regression در scikit-learn استفاده می­کنیم، مجبور نبودیم ویژگی­ها را استاندارد کنیم. با این حال، باید توجه داشت که تقریباً در تمام موارد دنیای واقعی استانداردسازی ضروری است.

**7.21 پیش‌بینی**

**مسئله**

شما می‌خواهید از یک شبکه عصبی برای پیش بینی استفاده کنید.

**راه‌حل**

از PyTorch برای ساخت یک شبکه عصبی پیشخور استفاده کنید، سپس با استفاده از فوروارد پیش‌بینی کنید:



بحث  
پیش بینی در PyTorch آسان است. هنگامی که شبکه عصبی خود را آموزش دادیم، می‌توانیم از روش فوروارد (که قبلاً به عنوان بخشی از فرآیند آموزش استفاده شده است) استفاده کنیم، که مجموعه‌ای از ویژگی‌ها را به عنوان ورودی می‌گیرد و یک گذر به جلو از شبکه انجام می‌دهد. در این راه حل، شبکه عصبی برای طبقه بندی باینری تنظیم شده است، بنابراین خروجی پیش بینی شده احتمال کلاس 1 بودن است. مشاهدات با مقادیر پیش بینی شده بسیار نزدیک به 1 به احتمال زیاد کلاس 1 هستند، در حالی که مشاهدات با مقادیر پیش بینی شده بسیار نزدیک به 0 هستند. به احتمال زیاد کلاس 0 هستند. بنا بر این، ما از روش گرد کردن برای تبدیل این مقادیر به 1 و 0 برای طبقه‌بندی کننده باینری خود استفاده می‌کنیم.

**8.21 مصورسازی تاریخچه‌ی آموزش**

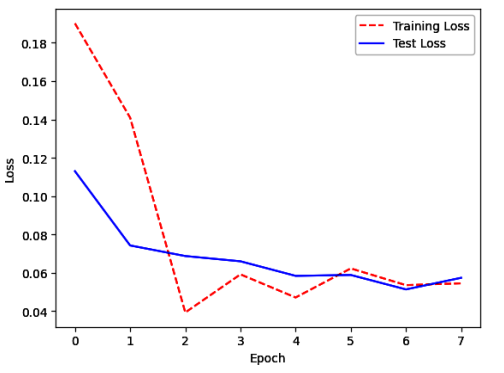
**مسئله**

شما می­خواهید «نقطه بهینه» را در میزان اشتباه (loss score) و یا دقت (accuracy score) شبکه عصبی پیدا کنید.

**راه‌حل**

از Matplotlib برای به تصویر کشیدن از دست دادن مجموعه آزمون و آموزشی در هر دوره استفاده کنید:





**بحث**

وقتی شبکه عصبی ما جدید باشد، عملکرد ضعیفی خواهد داشت. همانطور که شبکه عصبی از داده­های آموزشی یاد می­گیرد، خطای مدل در هر دو مجموعه آموزشی و آزمون کاهش می­یابد. با این حال، در یک نقطه خاص، شبکه عصبی شروع به «به خاطر سپردن[[7]](#footnote-7)» داده‌های آموزشی می­کند و بیش از حد برازش می­کند. هنگامی که این اتفاق می­افتد، خطای آموزش کاهش می­یابد در حالی که خطای آزمون شروع به افزایش می­کند. بنابراین، در بسیاری از موارد یک «نقطه شیرین» وجود دارد که در آن خطای آزمون (خطایی است که ما عمدتاً به آن اهمیت می‌دهیم) در پایین‌ترین نقطه خود قرار دارد. این تأثیر را می­توان به وضوح در راه‌حل ما مشاهده کرد که در آن خطای آموزش و آزمون را در هر دوره روی نمودار برده و رسم می‌کنیم. توجه داشته باشید که خطای آزمون در حوالی دوره 6 کمتر است، پس از آن میزان خطای آموزش همچنان کاهش می‌یابد در حالی که خطای آزمون شروع به افزایش می‌کند. از این مرحله به بعد، مدل بیش‌برازش کرده است.

**9.21 کاهش بیش‌برازش با منظم‌سازی**

**مسئله**

شما می­خواهید بیش‌برازش (overfit) را با تنظیم وزن شبکه خود کاهش دهید.

**راه‌حل**

سعی کنید پارامترهای شبکه را جریمه کنید که به آن منظم‌سازی وزن نیز گفته می‌شود:



بحث

یکی از راهبردهای مقابله با بیش‌برازش در شبکه‌های عصبی، جریمه کردن پارامترهای شبکه (مانند وزن‌ها) است، به‌گونه‌ای که به مقادیر کوچکتری گرایش پیدا کنند و مدلی ساده‌تر ایجاد شود که کمتر دچار بیش‌برازش شود. این روش به نام منظم‌سازی وزن یا تضعیف وزن شناخته می‌شود. به طور خاص، در منظم‌سازی وزن، یک جریمه به تابع خطا اضافه می‌شود، مانند نُرم L2.

در PyTorch، می‌توانیم منظم‌سازی وزن را با اضافه کردن weight\_decay=1e-5 در بهینه‌ساز اعمال کنیم، جایی که منظم‌سازی اتفاق می‌افتد. در این مثال، مقدار 1e-5 میزان جریمه اعمال‌شده بر مقادیر پارامترهای بزرگ‌تر را تعیین می‌کند. مقادیر بزرگ‌تر از 0 نشان‌دهنده‌ی منظم‌سازی L2 در PyTorch است.

**10.21 کاهش بیش‌برازش با توقف زودهنگام**

**مسئله**

می‌خواهید با توقف تمرین، تناسب بیش از حد را کاهش دهید، وقتی نمرات آموزش و آزمون شما متفاوت است.

**راه‌حل**

از Lightning در PyTorch که یک استراتژی به نام توقف زودهنگام است استفاده کنید:



**بحث**

همانطور که در دستور العمل 21.8 **بحث** کردیم، معمولاً در دوره­های آموزشی اول، خطاهای آموزش و آزمون کاهش می­یابد، اما در یک نقطه شبکه شروع به «به خاطر سپردن» داده­های آموزشی می­کند و باعث می­شود که خطای آموزشی کاهش یابد حتی در حالی که خطای آزمون شروع به افزایش می­کند. به دلیل این پدیده، یکی از متداول‌ترین و مؤثرترین روش­ها برای مقابله با بیش‌برازش ، نظارت بر روند آموزش مدل و توقف آموزش، درست در زمانی است که خطای آزمون شروع به افزایش می­کند. این استراتژی توقف زودهنگام نامیده می­شود.

در Keras، ما می­توانیم توقف زودهنگام را به عنوان یک تابع Callback پیاده‌سازی کنیم. Callback‌ ها توابعی هستند که می­توانند در مراحل خاصی از فرآیند آموزشی مانند پایان هر دوره، اعمال شوند.

با این حال، خود PyTorch برای شما یک کلاس توقف زودهنگام تعریف نمی‌کند، بنابراین در اینجا از کتابخانه محبوب lightening (که به عنوان PyTorch Lightning شناخته می‌شود) استفاده می‌کنیم تا از یک راه‌حل آماده استفاده کنیم. PyTorch Lightning یک کتابخانه سطح بالا برای PyTorch است که ویژگی‌های مفیدی را ارائه می‌دهد. در راه‌حل ما، از EarlyStopping(Pytorch Lightning) با پارامترهای monitor="val\_loss"، mode="min" و patience=3 استفاده کردیم تا مشخص کنیم که می‌خواهیم در هر دوره، ضرر آزمون (اعتبارسنجی) را زیر نظر داشته باشیم و اگر ضرر آزمون بعد از سه دوره (مقدار پیش‌فرض) بهبود نیافته باشد، آموزش متوقف شود.

اگر ما از callback EarlyStopping استفاده نمی‌کردیم، مدل برای حداکثر ۱۰۰۰ دوره بدون توقف به آموزش ادامه می‌داد.



**11.21 کاهش بیش‌برازش با Dropout**

**مسئله**

شما می­خواهید بیش‌برازش را کاهش دهید.

**راه‌حل**

با استفاده از dropout نویز را به معماری شبکه خود وارد کنید:



**بحث**

Dropout یک روش محبوب و قدرتمند برای منظم کردن شبکه­های عصبی است. در Dropout، هر بار که دسته­ای از مشاهدات برای آموزش ایجاد می­شود، نسبتی از واحدها در یک یا چند لایه در صفر ضرب می­شود (یعنی حذف می‌شود). در این تنظیمات، هر دسته در یک شبکه آموزش داده می‌شود (به عنوان مثال، پارامترهای یکسان)، اما هر دسته با نسخه‌ی کمی متفاوت‌تر از معماری آن شبکه مواجه می­شود.

Dropout موثر است زیرا با حذف مداوم و تصادفی واحدها در هر دسته، واحدها را مجبور می‌کند تا مقادیر پارامترهایی را که می‌توانند تحت طیف گسترده‌ای از معماری‌های شبکه عمل کنند، یاد بگیرند. به این معنی که آنها یاد می­گیرند که در برابر اختلالات (به عنوان مثال، نویز) در سایر واحدهای پنهان، مقاوم باشند و این مانع از حفظ داده­های آموزشی توسط شبکه می­شود.

امکان اضافه کردن ریزش به هر دو لایه‌ی مخفی و ورودی وجود دارد. هنگامی که یک لایه ورودی حذف می­شود، مقدار ویژگی آن برای آن دسته به شبکه وارد نمی­شود. یک انتخاب متداول برای قسمتی از واحدها که حذف می­شوند، 0.2 برای واحدهای ورودی و 0.5 برای واحدهای پنهان است.

در PyTorch، ما می توانیم dropout را با افزودن یک لایه nn.Dropout به معماری شبکه خود پیاده سازی کنیم. هر لایه nn.Dropout یک فراپارامتر تعریف شده توسط کاربر از واحدها را در لایه قبلی در هر دسته حذف می کند.

**12.21 ذخیره پیشرفت آموزش مدل**

**مسئله**

با توجه به یک شبکه عصبی که آموزش آن زمان زیادی طول می­کشد، می­خواهید پیشرفت خود را در صورت قطع شدن فرآیند آموزش ذخیره کنید.

**راه‌حل**

از تابع torch.save برای ذخیره مدل بعد از هر دوره استفاده کنید:



بحث

در دنیای واقعی، معمول است که شبکه های عصبی ساعت ها یا حتی روزها آموزش ببینند. در این مدت بسیاری از مشکلات ممکن است پیش بیایند: کامپیوترها ممکن است برق خود را از دست بدهند، سرورها از کار بیفتند یا دانشجویان فارغ التحصیل بی ملاحظه می توانند لپ تاپ شما را ببندند.

ما می توانیم از torch.save برای کاهش این مشکل با ذخیره مدل بعد از هر دوره استفاده کنیم. به طور خاص، پس از هر دوره، یک مدل را در مکان model.pt ذخیره می کنیم، و آرگومان دوم را در تابع torch.save. اگر فقط یک نام فایل (به عنوان مثال، model.pt) را وارد کنیم، آن فایل در هر دوره با آخرین مدل لغو می شود.

همانطور که می توانید تصور کنید، ما می توانیم هر چند دوره یکبار منطق اضافی برای ذخیره مدل معرفی کنیم، فقط در صورت کاهش ضرر یک مدل را ذخیره می‌کنیم. حتی می توانیم این رویکرد را با رویکرد توقف اولیه در PyTorch Lightning ترکیب کنیم تا مطمئن شویم که یک مدل را ذخیره می کنیم. مهم نیست که آموزش در چه دوره ای به پایان می رسد.

**13.21 تنظیم شبکه­های عصبی**

**مسئله**

شما می­خواهید به طور خودکار بهترین هایپرپارامترها را برای شبکه عصبی خود انتخاب کنید.

راه‌حل

از کتابخانه تنظیم ray در PyTorch استفاده کنید:



**بحث**

در دستورالعمل­های 12.1 و 12.2، ما استفاده از تکنیک­های انتخاب مدل scikit-learn برای شناسایی بهترین ابرپارامترهای یک مدل scikitlearn را پوشش دادیم. در دستور العمل 21.12 ما یاد گرفتیم که می­توانیم شبکه عصبی خود را بپیچانیم تا بتواند از API scikit-learn استفاده کند. در این دستور ما این دو تکنیک را برای شناسایی بهترین هایپرپارامترهای یک شبکه عصبی ترکیب می­کنیم.

ابرپارامترهای یک مدل، مهم هستند و باید با دقت انتخاب شوند. با این حال، قبل از اینکه به ذهنمان خطور کند که استراتژی‌های انتخاب مدل مانند جستجوی شبکه‌ای ایده‌ی خوبی است، باید بدانیم که اگر آموزش مدل ما معمولاً ۱۲ ساعت یا یک روز طول می‌کشید، این فرآیند جستجوی شبکه‌ای ممکن است یک هفته یا بیشتر طول بکشد. بنابراین، تنظیم خودکار فراپارامتر شبکه‌های عصبی، گلوله‌ی نقره‌ای نیست، اما در شرایط خاص ابزار مفیدی است.

در راه‌حل خود، ما یک جست و جوی شبکه‌ی اعتبارسنجی متقابل بر روی تعدادی گزینه برای الگوریتم بهینه سازی، تعداد دوره‌ها و اندازه‌ی دسته انجام دادیم. حتی اجرای این نمونه‌ی اسباب‌بازی چند دقیقه طول کشید، اما پس از انجام آن، می‌توانیم از best\_params\_ برای مشاهده هایپرپارامترهای شبکه عصبی با بهترین نتایج استفاده کنیم.

**14.21 مصورسازی شبکه­های عصبی**

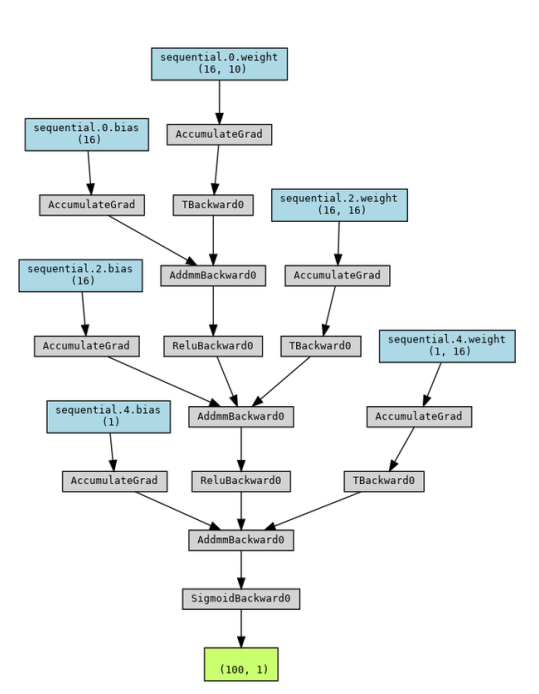
**مسئله**

شما می­خواهید به سرعت، معماری یک شبکه عصبی را به تصویر بکشید.

**راه‌حل**

از تابع make\_dot که در torch\_viz قرار دارد استفاده کنید:



اگر تصویر ذخیره شده در دستگاه خود را باز کنیم، می توانیم آن را مشاهده کنیم:

**بحث**

کتابخانه torchviz توابع کاربردی آسانی را برای تجسم سریع شبکه های عصبی ما و نوشتن آنها به عنوان تصویر فراهم می کند.

1. Multilayer Perceptron [↑](#footnote-ref-1)
2. - breaking the graph [↑](#footnote-ref-2)
3. Stochastic Gradient Descent (SGD) [↑](#footnote-ref-3)
4. Momentum [↑](#footnote-ref-4)
5. Root Mean Square Propagation (RMSProp) [↑](#footnote-ref-5)
6. Adaptive Moment Estimation (ADAM) [↑](#footnote-ref-6)
7. memorizing [↑](#footnote-ref-7)