**فصل 21 - شبکه‌های عصبی**

**21. 0 مقدمه**

در مرکز شبکه‌های عصبی، نورونی به نام گره یا نورون قرار دارد. گره، یک یا چند ورودی دریافت می‌کند، هر ورودی را در یک پارامتر دیگر(که به وزن نیز معروف است) ضرب می‌کند، مقادیر ورودی وزن‌دار را به همراه مقدار تعیین‌شده‌ای به نام بایاس (معمولاً 0) جمع می‌کند، سپس این مقدار به تابع فعال‌ساز وارد می‌شود. سپس این خروجی به سمت نورون‌های دیگر که در لایه‌های عمیق‌تر 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[[1]](#footnote-1) - که پرسپترون چندلایه[[2]](#footnote-2) نیز نامیده می‌شوند، ساده‌ترین شبکه‌های عصبی مصنوعی هستند که در هر محیط واقعی مورد استفاده قرار می‌گیرند. نام پیشخور از این واقعیت ناشی می‌شود 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 می‌دهد.

به طور خاص، شبکه‌های عصبی پیشخور شامل سه نوع لایه هستند. در ابتدای این شبکه عصبی، لایه ورودی قرار دارد که هر نورون شامل مقدار 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 نورون 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 پنهان[[3]](#footnote-3) قرار دارند. این لایه‌های پنهان 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 عصبی با لایه‌های پنهان زیاد (مثلاً 10، 100، 1000) شبکه‌های «عمیق» در نظر گرفته می‌شوند. آموزش شبکه‌های عصبی عمیق فرآیندی است که به عنوان «یادگیری عمیق» شناخته می‌شود.

شبکه‌های عصبی معمولاً با مقدار اولیه پارامترها به صورت مقادیر تصادفی کوچک از یک توزیع یکنواخت گاوسی یا عادی ایجاد می‌شوند. هنگامی که یک مشاهده (یا اغلب مجموعه‌ای از مشاهدات به نام دسته‌[[4]](#footnote-4)) از طریق شبکه آموزش داده می‌شود، مقدار خروجی با مقدار واقعی مشاهده با استفاده از تابع ضرر مقایسه می‌شود. به این روند انتشار روبه‌جلو[[5]](#footnote-5) می‌گویند. سپس یک الگوریتم از طریق شبکه «به عقب» می‌رود و مشخص می‌کند که هر پارامتر چقدر در خطای بین مقادیر پیش‌بینی شده و واقعی نقش داشته است، فرآیندی که به آن انتشار رو به عقب[[6]](#footnote-6) می‌گویند. به ازای هر پارامتر، الگوریتم بهینه‌سازی تعیین می‌کند که هر وزن چقدر باید برای بهبود خروجی تنظیم شود.

شبکه‌های عصبی با تکرار این فرآیند انتشار رو به جلو و انتشار به عقب برای هر نمونه در داده‌های آموزشی چندین بار یاد می‌گیرند (هر بار که تمام مشاهدات از طریق شبکه ارسال می‌شوند، دوره[[7]](#footnote-7) نامیده می‌شود و هر آموزش، معمولاً از چند دوره تشکیل می‌شود) و به ‌طور مکرر مقادیر پارامترها را با استفاده از فرآیندی به نام گرادیان کاهشی به آرامی به‌روزرسانی می‌کنند.

در این فصل، ما از همان کتابخانه‌ی پایتونی که در فصل آخر استفاده شده، PyTorch، برای ساخت، آموزش و ارزیابی انواع شبکه‌های عصبی استفاده خواهیم کرد. PyTorch یک ابزار محبوب در فضای یادگیری عمیق به دلیل API‌های خوب نوشته شده و نمایش بصری عملیات تانسور سطح پایین است که شبکه‌های عصبی را تقویت می‌کند. یکی از ویژگی‌های کلیدی این کتابخانه‌ PyTorch autograd نام دارد که به طور خودکار گرادیان‌های مورد استفاده برای بهینه‌سازی پارامترهای شبکه را پس از انتشار به جلو و انتشار به عقب محاسبه و ذخیره می‌کند.

شبکه‌های عصبی ایجاد شده با استفاده از کد PyTorch را می‌توان با استفاده از CPU (یعنی در لپ‌تاپ شما) و GPU (یعنی بر روی یک کامپیوتر یادگیری عمیق تخصصی) آموزش داد. در دنیای واقعی با داده‌های واقعی، اغلب لازم است شبکه‌های عصبی با استفاده از GPU آموزش داده شوند، زیرا فرآیند آموزش داده‌های بزرگ برای شبکه‌های پیچیده روی GPU به طور قابل توجهی سریع‌تر از CPU اجرا می‌شود. با این حال، تمام شبکه‌های عصبی موجود در این کتاب به اندازه کافی کوچک و ساده هستند که تنها در چند دقیقه با استفاده از CPU یک لپ‌تاپ آموزش داده شوند. فقط توجه داشته باشید که وقتی شبکه‌های بزرگ‌تر و داده‌های آموزشی بیشتری داریم، آموزش با استفاده از CPU به طور قابل توجهی کندتر از آموزش با استفاده از GPU است.

**1.21 استفاده از Autograd با PyTorch**

**مسئله**

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

**راه‌حل**

تنسورها 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 جهت‌دار غیرمدور[[8]](#footnote-8) 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 تنسورهای خود را بدون «شکستن نمودار»[[9]](#footnote-9) به آرایه‌های NumPy تبدیل کنیم و بالعکس، عبارتی که برای توصیف عملیات‌هایی که از autograd پشتیبانی نمی‌کنند استفاده می‌شود:



برای تبدیل این تنسور 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 detach کردن تنسور، از محاسبه خودکار گرادیان PyTorch جلوگیری می‌کند.

**همچنین، نگاه کنید به:**

* [آموزش PyTorch autograd](https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html)

**21.2 پیش‌پردازش داده برای شبکه‌های عصبی**

**مسئله**

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

**راه‌حل**

هر ویژگی را با استفاده از StandardScaler 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 بسیار ضعیف عمل می‌کنند. علاوه بر این، از آنجا که مقادیر ویژگی‌های یک مشاهده هنگام گذر از نورون‌های شبکه با هم ترکیب می‌شوند، اهمیت دارد که تمام ویژگی‌ها در یک مقیاس باشند.

با توجه به این دلایل، بهترین شیوه (اگرچه همیشه ضروری نیست؛ به عنوان مثال، زمانی که تمام ویژگی‌ها دودویی هستند) این است 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 StandardScaler در scikit-learn قابل انجام است.

با این حال، اگر بعد از ایجاد تنسورها با requires\_grad=True نیاز به انجام این عملیات دارید، باید این کار را به صورت بومی[[10]](#footnote-10) در PyTorch انجام دهید تا نمودار را نشکنید. در حالی که شما معمولاً ویژگی‌ها را قبل از شروع آموزش شبکه، استانداردسازی می‌کنید، ارزش این را دارد که بدانید چگونه همان کار را در PyTorch انجام دهید:



**3.21 طراحی یک شبکه عصبی**

**مسئله**

شما می­خواهید یک شبکه عصبی طراحی کنید.

**راه‌حل**

از کلاس nn.Module در PyTorch برای تعریف یک معماری شبکه عصبی ساده استفاده کنید:



شبکه‌های عصبی از لایه‌هایی متشکل از نورون‌ها 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 تحقیقاتی بسیاری از مقاله‌ها است. برای ساخت یک شبکه عصبی پیش‌خور در PyTorch ، باید در مورد معماری شبکه و فرآیند آموزش، تعدادی انتخاب انجام دهیم. به خاطر داشته باشید که هر نورون در لایه‌های پنهان:

1. تعدادی ورودی دریافت می‌کند.
2. هر ورودی را با مقدار یک پارامتر وزن‌دهی می‌کند.
3. تمام ورودی‌های وزن‌دار را با یک مقدار بایاس (معمولاً 0) جمع می‌کند.
4. بیشتر اوقات تابعی (که تابع فعال‌ساز نامیده می‌شود) را روی ورودی اعمال می‌کند.
5. خروجی را به یونیت‌ها یا همان گره‌ها در لایه‌ی بعدی ارسال می‌کند.

ابتدا، برای هر لایه در لایه‌های پنهان و خروجی، باید تعداد نورون‌هایی که در لایه قرار می‌دهیم و تابع فعال‌ساز را تعیین کنیم. به طور کلی، هر چه تعداد نورون‌ها در یک لایه بیشتر باشد، شبکه ما قادر به یادگیری الگوهای پیچیده‌تر خواهد بود. با این حال، تعداد بیشتر نورون‌ها ممکن است باعث بیش‌برازش شبکه به داده‌های آموزشی شده و به نحوی مخرب، بر عملکرد داده‌های آزمایشی تأثیر بگذارد. برای لایه‌های پنهان، تابع فعال‌ساز محبوب واحد یکسو شده خطی[[11]](#footnote-11) (ReLU) است:

که z مجموع ورودی‌های وزن‌دار و بایاس 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 دامنه‌ 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 پنهان 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 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[[12]](#footnote-12)

یک نورون با تابع فعال‌ساز سیگموئید.

طبقه‌بند چندکلاسی[[13]](#footnote-13)

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

رگرسیون

یک نورون 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 متقاطع دودویی[[14]](#footnote-14)

طبقه‌بند چندکلاسی

آنتروپی متقاطع رسته‌ای[[15]](#footnote-15)

رگرسیون

خطای میانگین مربعات[[16]](#footnote-16)

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[[17]](#footnote-17)، گرادیان کاهشی تصادفی با تکانه[[18]](#footnote-18)، پخش جذر میانگین مربعات[[19]](#footnote-19) و برآورد لحظه‌ای تطبیقی[[20]](#footnote-20) 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 فضای نام torch.nn.Module برای ایجاد یک شبکه عصبی ساده و ترتیبی استفاده می‌کنیم که می‌تواند طبقه‌بندی‌ دودویی انجام دهد. رویکرد استاندارد PyTorch برای این کار، ایجاد یک کلاس فرزند است که از کلاس torch.nn.Module ارث‌بری می‌کند، نمونه‌سازی یک معماری شبکه در تابع \_\_init\_\_ انجام می‌شود و تعریف عملیات ریاضی که می‌خواهیم بر روی هر عبور رو به جلو در کلاس انجام دهیم که در تابع forward نوشته می‌شود. 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، همچنین می‌توانیم این توابع فعال‌ساز 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 dense است (همچنین «کاملا متصل[[21]](#footnote-21)» نیز نامیده می‌شود)، به این معنی که تمام نورون‌های لایه قبلی به تمام نورون‌های لایه بعدی متصل هستند. در اولین لایه پنهان، out\_features=16 را تنظیم کردیم، به این معنی که لایه شامل 16 نورون است. این نورون‌ها دارای توابع فعال‌سازی ReLU هستند که در تابع forward در کلاس ما تعریف شده است:

x = nn.functional.relu(self.fc1(x)).

اولین لایه شبکه‌ی ما دارای اندازه‌ (16، 10) است، که به لایه اول می‌گوید که انتظار داشته باشد هر مشاهده از داده‌های ورودی ما دارای 10 مقدار ویژگی باشد. این شبکه برای طبقه‌بندی دودوسس طراحی شده است، بنابراین لایه خروجی تنها دارای یک نورون با تابع فعال‌ساز 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 ضرر برای طبقه‌بندی، ویکی‌پدیا](https://en.wikipedia.org/wiki/Loss_functions_for_classification)
* در مورد توابع خطا برای شبکه‌های عصبی عمیق در طبقه‌بندی

**4.21 آموزش یک طبقه‌بند دودویی**

**مسئله**

شما می‌خواهید یک شبکه عصبی طبقه‌بند دودویی آموزش دهید.

**راه‌حل**

از PyTorch برای ساخت یک شبکه عصبی روبه جلو استفاده کرده و آن را با استفاده از تابع fit آموزش دهید.

|  |
| --- |
| *# Import libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16, 1),  torch.nn.Sigmoid()  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet()  *# Define loss function, optimizer* criterion = nn.BCELoss() optimizer = RMSprop(network.parameters()) *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* epochs = 3 **for** epoch **in** range(epochs):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  print("Epoch:", epoch+1, "**\t**Loss:", loss.item())  *# Evaluate neural network* **with** torch.no\_grad():  output = network(x\_test)  test\_loss = criterion(output, y\_test)  test\_accuracy = (output.round() == y\_test).float().mean()  print("Test Loss:", test\_loss.item(), "**\t**Test Accuracy:",  test\_accuracy.item()) Epoch: 1 Loss: 0.19006995856761932 Epoch: 2 Loss: 0.14092367887496948  Epoch: 3 Loss: 0.03935524448752403  Test Loss: 0.06877756118774414 Test Accuracy: 0.9700000286102295 |

**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 شبکه‌ای است که در دستور العمل 21.3 وجود دارد (برای توضیح دقیق به آن دستورالعمل مراجعه کنید). تفاوت اینجاست که ما فقط شبکه‌ی عصبی را ایجاد کردیم. ما آن را آموزش ندادیم.

در انتها از with torch.no\_grad() برای ارزیابی شبکه استفاده می‌کنیم. این تابع می‌گوید که ما نباید گرادیان‌ها را برای عملیات تنسور انجام شده در این بخش از کد محاسبه کنیم. از آنجایی که ما از گرادیان‌ها فقط در طول فرآیند آموزش مدل استفاده می‌کنیم، نمی‌خواهیم گرادیان‌های جدیدی را برای عملیاتی که خارج از آن رخ می‌دهد (مانند پیش‌بینی یا ارزیابی) ذخیره کنیم.

متغیر epochs تعیین می‌کند که از چند دوره برای آموزش داده‌ها استفاده شود. batch\_size تعداد مشاهدات را برای انتشار در شبکه قبل از به روزرسانی پارامترها تنظیم می‌کند.

سپس تعداد دوره‌ها را تکرار می‌کنیم، با استفاده از تابع forward در شبکه رو به جلو حرکت می‌کنیم و سپس برای به‌روزرسانی گرادیان‌ها، حرکت‌های رو به عقب را انجام می‌دهیم. نتیجه یک مدل آموزش دیده است.

**5.21 آموزش یک طبقه‌بند چندکلاسه**

**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 تابع فعالساز softmax استفاده کنید:

|  |
| --- |
| *# Import libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split N\_CLASSES=3 EPOCHS=3  *# Create training and test sets* features, target = make\_classification(n\_classes=N\_CLASSES, n\_informative=9,  n\_redundant=0, n\_features=10, n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.nn.functional.one\_hot(torch.from\_numpy(target\_train).long(),  num\_classes=N\_CLASSES).float() x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.nn.functional.one\_hot(torch.from\_numpy(target\_test).long(),  num\_classes=N\_CLASSES).float()  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16,3),  torch.nn.Softmax()  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet() *# Define loss function, optimizer* criterion = nn.CrossEntropyLoss() optimizer = RMSprop(network.parameters())  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* **for** epoch **in** range(EPOCHS):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  print("Epoch:", epoch+1, "**\t**Loss:", loss.item())  *# Evaluate neural network* **with** torch.no\_grad():  output = network(x\_test)  test\_loss = criterion(output, y\_test)  test\_accuracy = (output.round() == y\_test).float().mean()  print("Test Loss:", test\_loss.item(), "**\t**Test Accuracy:",  test\_accuracy.item()) Epoch: 1 Loss: 0.8022041916847229 Epoch: 2 Loss: 0.775616466999054  Epoch: 3 Loss: 0.7751263380050659  Test Loss: 0.8105319142341614 Test Accuracy: 0.8199999928474426 |

**بحث**  
در این راه‌حل، ما یک شبکه عصبی مشابه طبقه‌بند دودویی از آخرین دستور، اما با تغییرات قابل توجه ایجاد کردیم. در داده‌های طبقه‌بندی‌ای که تولید کردیم، N\_CLASSES=3 را تنظیم کردیم. برای رسیدن به طبقه‌بندی چندکلاسه، ما همچنین از nn.CrossEntropyLoss() استفاده می‌کنیم که انتظار دارد هدف، کدگذاری تک روشن شود. برای انجام این کار، از تابع torch.nn.functional.one\_hot استفاده می‌کنیم و در نهایت منتهی می‌شویم به یک آرایه‌ رمزگذاری شده که موقعیت 1 آن کلاس یک مشاهده مشخص را نشان می‌دهد:



از آنجایی که این یک مسئله طبقه‌بندی چندکلاسه است، ما از یک لایه خروجی با اندازه 3 (یکی برای هر کلاس) استفاده کردیم که حاوی تابع فعال‌ساز softmax است. تابع فعال‌ساز 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 چندکلاسه 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 نورون خروجی که هیچ تابع فعال‌سازی ندارد استفاده کنید:

|  |
| --- |
| *# Import libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from sklearn.datasets import** make\_regression **from sklearn.model\_selection import** train\_test\_split EPOCHS=5  *# Create training and test sets* features, target = make\_regression(n\_features=10, n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1,1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1,1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16,1),  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet()  *# Define loss function, optimizer* criterion = nn.MSELoss() optimizer = RMSprop(network.parameters())  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* **for** epoch **in** range(EPOCHS):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  print("Epoch:", epoch+1, "**\t**Loss:", loss.item())  *# Evaluate neural network* **with** torch.no\_grad():  output = network(x\_test)  test\_loss = float(criterion(output, y\_test))  print("Test MSE:", test\_loss) Epoch: 1 Loss: 10764.02734375 Epoch: 2 Loss: 1356.510009765625 Epoch: 3 Loss: 504.9664306640625 Epoch: 4 Loss: 199.11314392089844 Epoch: 5 Loss: 191.20834350585938 Test MSE: 162.24497985839844 |

**بحث**  
ایجاد یک شبکه عصبی برای پیش‌بینی مقادیر پیوسته به جای احتمالات کلاس کاملاً امکان‌پذیر است. در مورد طبقه‌بند دودویی (دستور العمل 21.3) ما از یک لایه خروجی با یک نورون 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 فعال‌ساز 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 پیش‌بینی استفاده کنید.

**راه‌حل**

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

|  |
| --- |
| *# Import libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16, 1),  torch.nn.Sigmoid()  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet()  *# Define loss function, optimizer* criterion = nn.BCELoss() optimizer = RMSprop(network.parameters())  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* epochs = 3 **for** epoch **in** range(epochs):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  print("Epoch:", epoch+1, "**\t**Loss:", loss.item())  *# Evaluate neural network* **with** torch.no\_grad():  predicted\_class = network.forward(x\_train).round()  predicted\_class[0] Epoch: 1 Loss: 0.19006995856761932 Epoch: 2 Loss: 0.14092367887496948 Epoch: 3 Loss: 0.03935524448752403 tensor([1.]) |

بحث  
پیش بینی در PyTorch آسان است. هنگامی که شبکه عصبی خود را آموزش دادیم، می‌توانیم از تابع forward (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 در شبکه انجام می‌دهد. در این راه حل، شبکه عصبی برای طبقه بندی دودویی تنظیم شده است، بنابراین خروجی پیش‌بینی شده احتمال کلاس 1 بودن است. مشاهدات با مقادیر پیش‌بینی شده بسیار نزدیک به 1 به احتمال زیاد کلاس 1 هستند، در حالی که مشاهدات با مقادیر پیش‌بینی شده بسیار نزدیک به 0 به احتمال زیاد کلاس 0 هستند. بنابراین، ما از تابع round برای تبدیل این مقادیر به 1 و 0 برای طبقه‌بندی دودویی خود استفاده می‌کنیم.

**8.21 مصورسازی تاریخچه‌ آموزش**

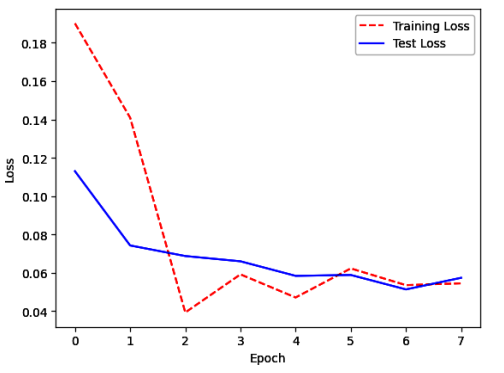
**مسئله**

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 مقدار خطای مجموعه آزمون و آموزشی در هر دوره استفاده کنید:

|  |
| --- |
| *# Load libraries* **import torch import torch.nn as nn from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split **import numpy as np import matplotlib.pyplot as plt**  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16, 1),  torch.nn.Sigmoid()  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet()  *# Define loss function, optimizer* criterion = nn.BCELoss() optimizer = RMSprop(network.parameters())  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* epochs = 8 train\_losses = [] test\_losses = [] **for** epoch **in** range(epochs):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  **with** torch.no\_grad():  train\_output = network(x\_train)  train\_loss = criterion(output, target)  train\_losses.append(train\_loss.item())  test\_output = network(x\_test)  test\_loss = criterion(test\_output, y\_test)  test\_losses.append(test\_loss.item())  *# Visualize loss history* epochs = range(0, epochs) plt.plot(epochs, train\_losses, "r--") plt.plot(epochs, test\_losses, "b-") plt.legend(["Training Loss", "Test Loss"]) plt.xlabel("Epoch") plt.ylabel("Loss") plt.show(); |



**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[[22]](#footnote-22)» 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 بهینه» 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 را با تنظیم وزن شبکه خود کاهش دهید.

**راه‌حل**

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

|  |
| --- |
| *# Import libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16, 1),  torch.nn.Sigmoid()  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet()  *# Define loss function, optimizer* criterion = nn.BCELoss() optimizer = torch.optim.Adam(network.parameters(), lr=1e-4, weight\_decay=1e-5)  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* epochs = 100 **for** epoch **in** range(epochs):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  *# Evaluate neural network* **with** torch.no\_grad():  output = network(x\_test)  test\_loss = criterion(output, y\_test)  test\_accuracy = (output.round() == y\_test).float().mean()  print("Test Loss:", test\_loss.item(), "**\t**Test Accuracy:",  test\_accuracy.item()) Test Loss: 0.4030887186527252 Test Accuracy: 0.9599999785423279 |

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، بیش‌برازش را کاهش دهید، وقتی امتیازهای آموزش و آزمون شما از هم فاصله می‌گیرد.

**راه‌حل**

از Lightning در PyTorch برای پیاده‌سازی استراتژی توقف زودهنگام استفاده کنید:

|  |
| --- |
| *# Import libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **import lightning as pl from lightning.pytorch.callbacks.early\_stopping import** EarlyStopping **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16, 1),  torch.nn.Sigmoid()  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  **class LightningNetwork**(pl.LightningModule):  **def** \_\_init\_\_(self, network):  super().\_\_init\_\_()  self.network = network  self.criterion = nn.BCELoss()  self.metric = nn.functional.binary\_cross\_entropy  **def** training\_step(self, batch, batch\_idx):  *# training\_step defines the train loop.*  data, target = batch  output = self.network(data)  loss = self.criterion(output, target)  self.log("val\_loss", loss)  **return** loss  **def** configure\_optimizers(self):  **return** torch.optim.Adam(self.parameters(), lr=1e-3)  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Initialize neural network* network = LightningNetwork(SimpleNeuralNet())  *# Train network* trainer = pl.Trainer(callbacks=[EarlyStopping(monitor="val\_loss", mode="min",  patience=3)], max\_epochs=1000) trainer.fit(model=network, train\_dataloaders=train\_loader) GPU available: False, used: False TPU available: False, using: 0 TPU cores IPU available: False, using: 0 IPUs HPU available: False, using: 0 HPUs | Name | Type | Params ---------------------------------------------- 0 | network | SimpleNeuralNet | 465 1 | criterion | BCELoss | 0 ---------------------------------------------- 465 Trainable params 0 Non-trainable params 465 Total params 0.002 Total estimated model params size (MB) /usr/local/lib/python3.10/site-packages/lightning/pytorch/trainer/  connectors/data\_connector.py:224: PossibleUserWarning:  The dataloader, train\_dataloader, does not have many workers which  may be a bottleneck. Consider increasing the value of the `num\_workers`  argument (try 7 which is the number of cpus on this machine)  in the `DataLoader` init to improve performance.  rank\_zero\_warn( /usr/local/lib/python3.10/site-packages/lightning/pytorch/trainer/  trainer.py:1609: PossibleUserWarning: The number of training batches (9)  is smaller than the logging interval Trainer(log\_every\_n\_steps=50).  Set a lower value for log\_every\_n\_steps if you want to see logs  for the training epoch.  rank\_zero\_warn( Epoch 23: 100%|███████████████| 9/9 [00:00<00:00, 59.29it/s, loss=0.147, v\_num=5] |

**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.

درPyTorch ، ما می­توانیم توقف زودهنگام را به عنوان یک تابع Callback پیاده‌سازی کنیم. Callback‌‌ها 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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *# Train network* trainer = pl.Trainer(max\_epochs=1000) trainer.fit(model=network, train\_dataloaders=train\_loader) GPU available: False, used: False TPU available: False, using: 0 TPU cores IPU available: False, using: 0 IPUs HPU available: False, using: 0 HPUs | Name | Type | Params ---------------------------------------------- 0 | network | SimpleNeuralNet | 465 1 | criterion | BCELoss | 0 ---------------------------------------------- 465 Trainable params 0 Non-trainable params 465 Total params 0.002 Total estimated model params size (MB)   |  |  | | --- | --- | | Epoch 999: 100%|████████████| 9/9 [00:01<00:00,  v\_num=6] `Trainer.fit` stopped: `max\_epochs=1000` reached. Epoch 999: 100%|████████████| 9/9 [00:01<00:00, | 7.95it/s, loss=0.00188, | | 7.80it/s, loss=0.00188, |  |   v\_num=6] |

**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:

|  |
| --- |
| *# Load libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16, 1),  torch.nn.Dropout(0.1), *# Drop 10% of neurons*  torch.nn.Sigmoid(),  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet()  *# Define loss function, optimizer* criterion = nn.BCELoss() optimizer = RMSprop(network.parameters())  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* epochs = 3 **for** epoch **in** range(epochs):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  print("Epoch:", epoch+1, "**\t**Loss:", loss.item())  *# Evaluate neural network* **with** torch.no\_grad():  output = network(x\_test)  test\_loss = criterion(output, y\_test)  test\_accuracy = (output.round() == y\_test).float().mean()  print("Test Loss:", test\_loss.item(), "**\t**Test Accuracy:",  test\_accuracy.item()) Epoch: 1 Loss: 0.18791493773460388 Epoch: 2 Loss: 0.17331615090370178  Epoch: 3 Loss: 0.1384529024362564  Test Loss: 0.12702330946922302 Test Accuracy: 0.9100000262260437 |

**بحث**

Dropout یک روش محبوب و قدرتمند برای منظم‌سازی شبکه­های عصبی است. در dropout، هر بار که دسته­ای از مشاهدات برای آموزش ایجاد می­شود، نسبتی از نورون‌ها 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 نورون‌ها در هر دسته، نورون‌ها 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 نورون‌های پنهان، مقاوم باشند و این مانع از حفظ داده­های آموزشی توسط شبکه می­شود.

امکان اضافه کردن dropout به هر دو لایه‌ی پنهان 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 نورون‌ها که حذف می­شوند، 0.2 برای نورون‌های ورودی و 0.5 برای نورون‌های پنهان است.

در PyTorch، ما می‌توانیم dropout را با افزودن یک لایه nn.Dropout به معماری شبکه خود پیاده‌سازی کنیم. هر لایه nn.Dropout یک ابرپارامتر تعریف شده توسط کاربر از نورون‌ها را از لایه قبلی در هر دسته حذف می‌کند.

**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:

|  |
| --- |
| *# Load libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16, 1),  torch.nn.Dropout(0.1), *# Drop 10% of neurons*  torch.nn.Sigmoid(),  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet()  *# Define loss function, optimizer* criterion = nn.BCELoss() optimizer = RMSprop(network.parameters())  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* epochs = 5 **for** epoch **in** range(epochs):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  *# Save the model at the end of every epoch*  torch.save(  {  'epoch': epoch,  'model\_state\_dict': network.state\_dict(),  'optimizer\_state\_dict': optimizer.state\_dict(),  'loss': loss,  },  "model.pt"  )  print("Epoch:", epoch+1, "**\t**Loss:", loss.item()) Epoch: 1 Loss: 0.18791493773460388 Epoch: 2 Loss: 0.17331615090370178 Epoch: 3 Loss: 0.1384529024362564 Epoch: 4 Loss: 0.1435958743095398 Epoch: 5 Loss: 0.1 |

بحث

در دنیای واقعی، معمول است که شبکه‌های عصبی ساعت‌ها 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 فارغ‌التحصیل بی ملاحظه می‌توانند لپ تاپ شما را ببندند.

ما می‌توانیم 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 می‌کنیم، که آرگومان دوم تابع torch.save است. اگر فقط یک نام فایل (به عنوان مثال، model.pt) وارد کنیم، آن فایل در هر دوره با آخرین مدل بازنویسی می‌شود.

همانطور که می‌توانید تصور کنید، ما می‌توانیم هر چند دوره یک بار منطق اضافی برای ذخیره مدل معرفی کنیم، مثلا اینکه فقط در صورت کاهش خطا یک مدل را ذخیره کنیم یا هر منطق دیگری. حتی می‌توانیم این رویکرد را با رویکرد توقف زودهنگام در PyTorch Lightning ترکیب کنیم تا مطمئن شویم که یک مدل را ذخیره می‌کنیم بدون توجه به اینکه آموزش در چه دوره‌ای به پایان می‌رسد.

**13.21 تنظیم شبکه­های عصبی**

**مسئله**

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

**راه‌حل**

از کتابخانه تنظیم ray در PyTorch استفاده کنید:

|  |
| --- |
| *# Load libraries* **from functools import** partial **import numpy as np import os import torch import torch.nn as nn import torch.nn.functional as F import torch.optim as optim from torch.optim import** RMSprop **from torch.utils.data import** random\_split, DataLoader, TensorDataset **from ray import** tune **from ray.tune import** CLIReporter **from ray.tune.schedulers import** ASHAScheduler **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using `Sequential`* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self, layer\_size\_1=10, layer\_size\_2=10):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, layer\_size\_1),  torch.nn.ReLU(),  torch.nn.Linear(layer\_size\_1, layer\_size\_2),  torch.nn.ReLU(),  torch.nn.Linear(layer\_size\_2, 1),  torch.nn.Sigmoid()  )  **def** forward(self, x):  x = self.sequential(x)  **return** x config = {  "layer\_size\_1": tune.sample\_from(**lambda** \_: 2 \*\* np.random.randint(2, 9)),  "layer\_size\_2": tune.sample\_from(**lambda** \_: 2 \*\* np.random.randint(2, 9)),  "lr": tune.loguniform(1e-4, 1e-1), }  scheduler = ASHAScheduler(  metric="loss",  mode="min",  max\_t=1000,  grace\_period=1,  reduction\_factor=2 )  reporter = CLIReporter(  parameter\_columns=["layer\_size\_1", "layer\_size\_2", "lr"],  metric\_columns=["loss"] )  *# Train neural network* **def** train\_model(config, epochs=3):  network = SimpleNeuralNet(config["layer\_size\_1"], config["layer\_size\_2"])  criterion = nn.BCELoss()  optimizer = optim.SGD(network.parameters(), lr=config["lr"], momentum=0.9)  train\_data = TensorDataset(x\_train, y\_train)  train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer*  network = torch.compile(network)  **for** epoch **in** range(epochs):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  tune.report(loss=(loss.item()))  result = tune.run(  train\_model,  resources\_per\_trial={"cpu": 2},  config=config,  num\_samples=1,  scheduler=scheduler,  progress\_reporter=reporter )  best\_trial = result.get\_best\_trial("loss", "min", "last") print("Best trial config: {}".format(best\_trial.config)) print("Best trial final validation loss: {}".format(  best\_trial.last\_result["loss"])) best\_trained\_model = SimpleNeuralNet(best\_trial.config["layer\_size\_1"],  best\_trial.config["layer\_size\_2"]) == Status == Current time: 2023-03-05 23:31:33 (running for 00:00:00.07) Memory usage on this node: 1.7/15.6 GiB Using AsyncHyperBand: num\_stopped=0 Bracket: Iter 512.000: None | Iter 256.000: None | Iter 128.000: None |  Iter 64.000: None | Iter 32.000: None | Iter 16.000: None |  Iter 8.000: None | Iter 4.000: None | Iter 2.000: None |  Iter 1.000: None Resources requested: 2.0/7 CPUs, 0/0 GPUs, 0.0/8.95 GiB heap,  0.0/4.48 GiB objects Result logdir: /root/ray\_results/train\_model\_2023-03-05\_23-31-33 Number of trials: 1/1 (1 RUNNING) ... |

**بحث**

در دستورالعمل­های 12.1 و 12.2، ما استفاده از تکنیک­های انتخاب مدل scikit-learn برای شناسایی بهترین ابرپارامترهای یک مدل scikit-learn را پوشش دادیم. اگرچه به صورت کلی از روش‌های scikit-learn می‌توان برای شبکه‌های عصبی نیز استفاده کرد، با این‌حال کتابخانه ray یک API حرفه‌ای را در اختیار ما قرار می‌دهد که به ما اجازه می‌دهد تا پیاده‌سازی‌ها را روی هر دو واحد پردازشی CPU و GPU زمانبندی کنیم.

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 ابرپارامتر شبکه‌های عصبی، گلوله‌ی نقره‌ای نیست، اما در شرایط خاص ابزار مفیدی است.

در راه‌حل خود، ما یک جستجوی شبکه‌ی اعتبارسنجی متقابل بر روی تعدادی گزینه برای الگوریتم بهینه‌سازی، 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 ابرپارامترهای شبکه عصبی با بهترین نتایج استفاده کنیم.

**14.21 مصورسازی شبکه­های عصبی**

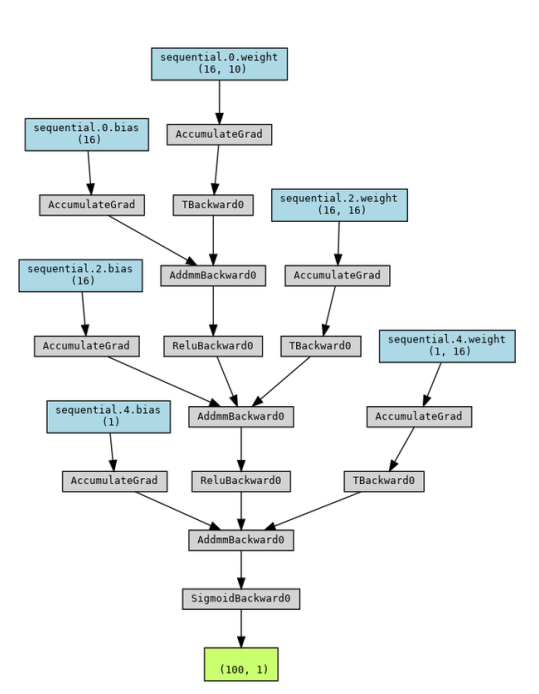
**مسئله**

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:

|  |
| --- |
| *# Load libraries* **import torch import torch.nn as nn import numpy as np from torch.utils.data import** DataLoader, TensorDataset **from torch.optim import** RMSprop **from torchviz import** make\_dot **from sklearn.datasets import** make\_classification **from sklearn.model\_selection import** train\_test\_split  *# Create training and test sets* features, target = make\_classification(n\_classes=2, n\_features=10,  n\_samples=1000) features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.1, random\_state=1)  *# Set random seed* torch.manual\_seed(0) np.random.seed(0)  *# Convert data to PyTorch tensors* x\_train = torch.from\_numpy(features\_train).float() y\_train = torch.from\_numpy(target\_train).float().view(-1, 1) x\_test = torch.from\_numpy(features\_test).float() y\_test = torch.from\_numpy(target\_test).float().view(-1, 1)  *# Define a neural network using Sequential* **class SimpleNeuralNet**(nn.Module):  **def** \_\_init\_\_(self):  super(SimpleNeuralNet, self).\_\_init\_\_()  self.sequential = torch.nn.Sequential(  torch.nn.Linear(10, 16),  torch.nn.ReLU(),  torch.nn.Linear(16,16),  torch.nn.ReLU(),  torch.nn.Linear(16, 1),  torch.nn.Sigmoid()  )  **def** forward(self, x):  x = self.sequential(x)  **return** x  *# Initialize neural network* network = SimpleNeuralNet()  *# Define loss function, optimizer* criterion = nn.BCELoss() optimizer = RMSprop(network.parameters())  *# Define data loader* train\_data = TensorDataset(x\_train, y\_train) train\_loader = DataLoader(train\_data, batch\_size=100, shuffle=**True**)  *# Compile the model using torch 2.0's optimizer* network = torch.compile(network)  *# Train neural network* epochs = 3 **for** epoch **in** range(epochs):  **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  optimizer.zero\_grad()  output = network(data)  loss = criterion(output, target)  loss.backward()  optimizer.step()  make\_dot(output.detach(), params=dict(  list(  network.named\_parameters()  ) )).render(  "simple\_neural\_network",  format="png" )  'simple\_neural\_network.png' |

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

**بحث**

کتابخانه torchviz توابع کاربردی آسانی را برای تصویرسازی سریع شبکه‌های عصبی ما و ذخیره آنها به عنوان تصویر فراهم می‌کند.

1. feedforward [↑](#footnote-ref-1)
2. Multilayer Perceptron [↑](#footnote-ref-2)
3. Hidden Layers [↑](#footnote-ref-3)
4. Batch [↑](#footnote-ref-4)
5. Forward Propagation [↑](#footnote-ref-5)
6. Back Propagation [↑](#footnote-ref-6)
7. epoch [↑](#footnote-ref-7)
8. Directed Acyclic Graph (DAG) [↑](#footnote-ref-8)
9. - breaking the graph [↑](#footnote-ref-9)
10. natively [↑](#footnote-ref-10)
11. Rectified Linear Unit (ReLU) [↑](#footnote-ref-11)
12. Binary classification [↑](#footnote-ref-12)
13. Multiclass classification [↑](#footnote-ref-13)
14. Binary cross-entropy [↑](#footnote-ref-14)
15. Categorical cross-entropy [↑](#footnote-ref-15)
16. Mean square error [↑](#footnote-ref-16)
17. Stochastic Gradient Descent (SGD) [↑](#footnote-ref-17)
18. Momentum [↑](#footnote-ref-18)
19. Root Mean Square Propagation (RMSProp) [↑](#footnote-ref-19)
20. Adaptive Moment Estimation (ADAM) [↑](#footnote-ref-20)
21. Fully Connected [↑](#footnote-ref-21)
22. memorizing [↑](#footnote-ref-22)