فصل ۵

شبکه های عصبی پیچشی - Pytorch

Pytorch چیست؟

پایتورچ، سیستمی برای اجرای محاسباتِ پویای گرافی، بر روی تنسورهایی است که رفتار آنها، مشابه رفتارِ ndarray ها در numpy می‌باشد. پایتورچ یک موتورِ خودکار مشتقیِ قدرتمند دارد که نیاز به پیاده‌سازی دستیِ انتشارِ پس‌رو [۱] را از بین می‌برد.
برای یادگیریِ پایتورچ می‌توانید از این لینک استفاده کنید یا از ریز مستندات، در API doc بهره ببرید.
در این جلسه در ۵ قسمت، قصد داریم شما را با سه سطح انتزاعی مختلف در پایتورچ آشنا کنیم. در قسمت اول، داده‌های خود را برای استفاده در پایتورچ آماده‌سازی خواهیم کرد. در قسمت دوم، بر روی پایین‌ترین سطحِ تنسورهای پایتورچ (که به آن پایتورچِ خالص نیز می‌گوییم) کار خواهیم کرد. در قسمت سوم از Pytorch Module API برای تعریف کردنِ معماری دلخواهِ شبکه‌های عصبی، بهره خواهیم برد. در قسمت چهارم از Pytorch Sequential API برای تعریف آسانِ یک شبکه‌ی پیش‌خور [۲] خطیِ استفاده خواهیم کرد و در قسمت پنجم نیز شما آزاد خواهید بود که با بهره بردن از هرگونه لایه یا بهینه‌ساز و استفاده و تغییرِ ابَرپارامترها یا دیگر ویژگی ها، شبکه‌ای را بسازید که بر روی مجموعه داده‌ی CIFAR-10 دقت قابل قبولی را بدست بیاورید. در ادامه، جدولی را برای مقایسه‌ی سه روشی که تعریف کردیم، خواهید دید:

API	انعطاف‌پذیری	راحتی
خالص	زیاد	کم
nn.Module	زیاد	متوسط
nn.Sequential	کم	زیاد

GPU

شما می‌توانید بصورت دستی در Colab با انتخابِ `GPU` تحت قسمتِ Gardware Accelerator در منوی Runtime->Change Runtime type ، به یک دستگاهِ GPU منتقل شوید. این کار باید قبل از اجرای کدهای ذیل قسمتِ GPU در فایلِ ژوپیتر مربوط به این جلسه، صورت بپذیرد.

۱.۵ تمرین اول

در این قسمت قصد داریم که یک شبکه عصبی ساده برای تشخیص داده‌های موجود بر یک سهمی ساده که مقدار متغیر مستقل آن یک بازه حاوی ۱۰۰ عدد بین ۱ تا ۱- است.

[۱] backpropagation
[۲] feed-forward

۱.۱.۵ پایتورچِ خالص

PyTorch با API های سطح بالا به ما کمک می کند تا معماری های مدل را به راحتی تعریف کنیم ، که در قسمت سوم این آموزش به آنها می‌پردازیم. در این قسمت، برای یادگرفتنِ هر چه بهترِ موتورِ خودکار ، با عناصر پایتورچ خالص کار خود را آغاز خواهیم کرد. بعد از این تمرین، شما دلیل استفاده از API های سطح بالا را خواهید فهمید. ما با یک شبکه‌ی ساده‌ی تمام متصلِ ReLU با یک لایه‌ی پنهان و بدونِ بایاس، برای پیشبینی مقدار تقریبی متغیر بر روی سهمی شروع خواهیم کرد. این پیاده‌سازی، مرحله‌ی گذرِ پیش‌رو [۱] را با استفاده از عملیاتی بر روی تنسورهای پایتورچ، محاسبه می‌کند و از autograd پایتورچ برای محاسبه‌ی گرادیان ها، بهره می‌برد. شما باید هر خط از کد را متوجه شوید زیرا در ادامه، یک نسخه سخت‌تر را باید بنویسید.
هنگامی که یک تنسور پایتورچ را با `requires_grad=True` می‌سازیم، عملیاتی که آن تنسور در آنها دخیل است، فقط مقادیر را محاسبه نمیکنند، بلکه یک گراف محاسباتی نیز در پس‌زمینه ایجاد می‌کنند که به ما اجازه می‌دهد به سادگی انتشارِ پس‌رو در طول گراف، برای محاسبه‌ی گرادیان برخی از تنسورها با توجه به یک خطای کاهشی، انجام دهیم. به طور مشخص، اگر x یک تنسور با `x.requires_grad==True` باشد، آنگاه بعد از انتشارِ پس‌رو، `x.grad` تنسور دیگری خواهد بود که گرادیانِ x را با توجه به مقدار اسکالر خطا در پایان، نگهداری می‌کند.

۲.۱.۵ Pytorch Module API

پایتورچِ خالص نیاز دارد که ما تمامی پارامترهای تنسورها را بصورت دستی، پیگیری کنیم. این مورد برای شبکه های کوچک با تعداد کمی تنسور، اشکالی ندارد، اما ردیابی ده ها یا صدها تنسور در شبکه های بزرگ بسیار ناخوشایند و مستعد خطا است.
پایتورچ با تعریف کردنِ nn.Module API زمینه‌ی ساخت ممعاری های دلخواه شبکه در کنار قابلیت ردیابی خودکار تمام پارامترهای قابل یادگیری را برای ما فراهم کرده است. در قسمت دوم این جلسه، ما SGD را تعریف کردیم. پایتورچ همچنین پکیج `torch.optim` را که تمامی بهینه‌سازهای معمولی مانند RMSProp، Adagradو Adam پیاده سازی کرده، فراهم کرده است و حتی از روشهای تقریبی مرتبه دوم مانند L-BFGS پشتیبانی می کند . برای استفاده کردن از Module API قدم‌های زیر را باید پی بگیریم:

- در زیرکلاسِ `nn.Module` به کلاس شبکه خود یک نامِ شهودی مانند `Net` بدهید.
- در سازنده‌ی [۲] `__init__()` تمامی لایه‌های مورد نیاز خود را به عنوانِ ویژگی‌های کلاس [۳] تعریف کنید. لایه‌های شیءای مانند `nn.Linear` خود زیرکلاس هایی از `nn.Module` هستند و درون خود پارامترهای یادگیری دارند. پس شما نیازی به تعریف و مقداردهی تنسورهای خام ندارید. `nn.Module` این پارامترهای داخلی را خود، ردیابی می‌کند. برای یادگیری بیشترِ لایه‌های از پیش طراحی شده می‌توانید از این لینک بهره ببرید.
  هشدار: فراموش نکنید که ابتدا `super().__init__()` را فراخوانی کنید.
- در تابعِ `forward()` ، *اتصال* [۴] شبکه خود را تعریف کنید. شما باید از ویژگی‌های تعریف شده داخلِ `__init__` به عنوان فراخوانی توابعی که تنسورها را به عنوان ورودی میگیرند و تنسورهای " تبدیل شده [۵] " استفاده کنید. در `forward()` هیچ لایه‌ای با پارامترهای قابل یادگیری تعریف نکنید. همه‌ی این پارامترها باید از قبل در `__init__` تعریف شوند.

بعد از آنکه زیرکلاس های ماژول خود را تعریف کردید، می‌توانید آن را به عنوان یک شیء تعریف کنید و درست مانند NN forward در قسمت دو، آن را فراخوانی کنید.

---
[۱] forward pass
[۲] constructor
[۳] class attributes
[۴] connectivity
[۵] transformed

۲.۵ تمرین دوم

۱.۲.۵ توزیع نرمال

توزیع گوسی [۱] یا توزیع نرمال، یکی از مهم‌ترین و پراستفاده‌ترین توزیع‌های احتمالی در آمار و احتمالات است. توزیع گوسی به خوبی ویژگی‌های بسیاری از پدیده‌ها و داده‌های واقعی را توصیف می‌کند و در مختلف زمینه‌های علمی و مهندسی بسیار مفید است. برای اطلاعات بیشتر به این لینک مراجعه کنید.

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{۱.۵}$$

۲.۲.۵ Pytorch Sequential API

در قسمت قبل، شما با pytorch Module API ، که به ماامکانِ تعریف لایه‌های قابل یادگیری دلخواه و اتصالات آنها را می‌دهد، آشنا شدید.
برای مدلهای ساده مانند دسته‌ای از لایه‌های پیش‌خور [۲] شما هنوز نیاز دارید که سه مرحله را بگذرانید: زیرکلاسِ `nn.Module` ، اختصاص دادن لایه‌ها به ویژگیهای کلاس در `__init__` و سپس فراخوانی تک به تکِ هر لایه در `forward()` . سوال اینجاست که آیا راهِ راحت‌تری وجود دارد؟
خوشبختانه پایتورچ یک container Module به نامِ `nn.Sequential` را ارائه می‌دهد تمامی مراحلی که ذکر کردیم را با هم ادغام، و آن‌ها را به یک مرحله تبدیل می‌کند. این ماژول به انعطاف‌پذیریِ `nn.Module` نیست زیرا شما قادر به تعریفِ یک توپولوژی پیچیده‌تر از تعدادیِ پیش‌خور، نخواهید بود، اما برای بسیاری از موارد، همین ماژول، کافی خواهد بود.

۳.۲.۵ اجرای یک شبکه Sequential

در این قسمت در متغیر model یک شبکه عصبی Sequential را با توجه به شبکه خواسته شده ایجاد کنید، سپس با توجه به توابع درج شده میزان خطا و نحوه پیشبینی را مشخص کنید.

۳.۵ تمرین سوم

۱.۳.۵ آماده‌سازی

حال می‌خواهیم مجموعه داده‌ی Dataset را بارگذاری [۳] کنیم. وقتی برای بار اول این کار را می‌کنید، ممکن است چند دقیقه زمان ببرد.
پایتورچ برای پیش‌پردازش و گذر از بین مینی دسته‌ها [۴] ابزارهای بسیار مناسبی را در اختیار ما قرار داده است.
کد این قسمت را می‌توانید در فایل این جلسه، ذیل بخشِ Loading Dataset مشاهده کنید.

۲.۳.۵ پایتورچِ خالص - شبکه‌ی پیچشیِ دو لایه

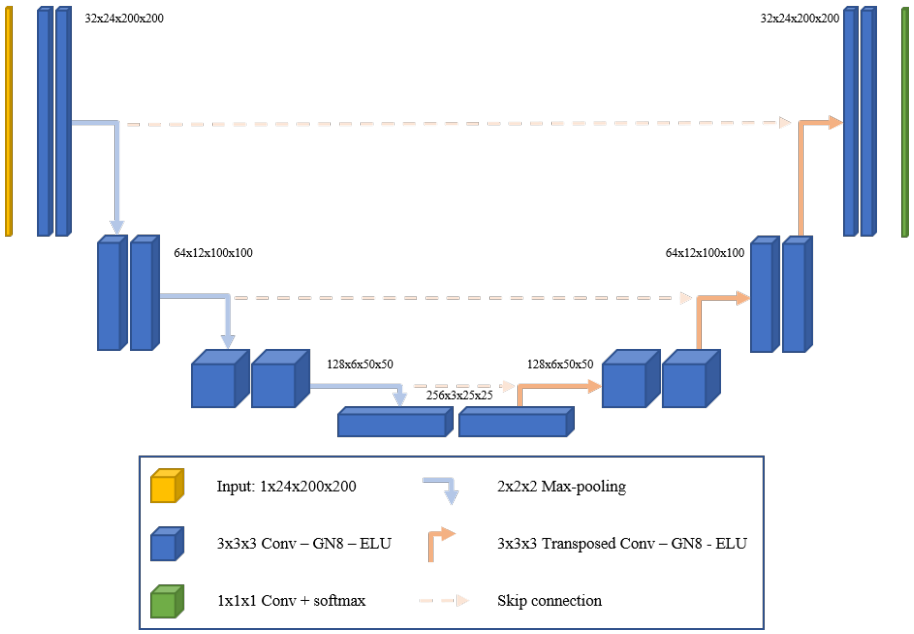
در این قسمت ، در فایل مربوط به این جلسه، شما پیاده‌سازیِ تابعِ `Convblock` را تکمیل خواهید کرد این تابع یک گذرِ پیش‌رو بر روی یک شبکه پیچشیِ دو لایه اجرا می‌کند. مانند قسمت قبل ما می‌توانیم پیاده‌سازی خود را با پاس دادن مقادیر صفر در طول شبکه، تست کنیم. شبکه باید معماری‌ای به صورتی که در ذیل می‌آید، داشته باشد:

[۱] Gaussian distribution
[۲] feed-forward
[۳] load
[۴] minibatch

- یک لایه‌ی پیچشی با فیلترهای `input_channel` ، که هر کدام دارای شکلِ kernel x kernel هستند و یک لایه‌گذاریِ [۱] به اندازه به اندازه padding و گام‌هایی [۲] به اندازه stride
- ReLU غیر خطی
- یک لایه Batch Normalization برای جلوگیری از بیش‌برازش و افزایش پایداری
- یک لایه‌ی پیچشی با فیلترهای `output_channel` ، که هر کدام دارای شکلِ kernel x kernel هستند و یک لایه‌گذاریِ [۳] به اندازه به اندازه padding و گام‌هایی به اندازه stride
- ReLU غیرخطی

U-Net ۳.۳.۵

شبکه U-Net یک مدل عمیق کانوولوشنی برای تشخیص و بازسازی اشیاء در تصاویر است. این مدل از معماری U-شکل تشکیل شده است که شامل بخش کدگذار برای استخراج ویژگی‌های تصویر و بخش کدگشا برای بازسازی تصویر از این ویژگی‌ها می‌باشد. شبکه U-Net از لایه‌های کانوولوشنی و پیوندها بین بخش‌های مختلف برای بهبود عملکرد در تشخیص اشیاء در تصاویر استفاده می‌کند. این مدل به خوبی در حوزه‌های تصویرپردازی و پزشکی برای تشخیص تومورها، تشخیص سلول‌های آمیخته، و دیگر کاربردهای مشابه مورد استفاده قرار می‌گیرد. برای اطلاعات بیشتر به این لینک مراجعه کنید. در قسمت زیر به بررسی لایه های مختلف این شبکه می‌پردازیم:



شکل ۱.۵: ساختار شبکهU-Net

کدگذار [۴] ۴.۳.۵

بخش کدگذار در سمت چپ معماری U-Net واقع شده است. این بخش مسئول استخراج ویژگی‌ها از تصویر ورودی می‌باشد. در این قسمت کدگذار از ۴ بلاک Convblock که در قسمت قبل ایجاد کردیم تشکیل شده است. این لایه‌ها

[۱] padding
[۲] stride
[۳] padding
[۴] Encoder

به تدریج ابعاد مکانی تصویر ورودی را کاهش می‌دهند و تعداد نقشه‌های ویژگی را افزایش می‌دهند. تعداد فیلترهای این شبکه برای این ۴ لایه به صورت زیر است:

- ورودی: `input_channel`
- بلاک اول: `32x`
- بلاک دوم: `64x`
- بلاک سوم: `128x`

۵.۳.۵ گلوگاه [۱]

لایه یا لایه‌های وسطی به عنوان گلوگاه شناخته می‌شوند و وظیفه اتصال کدگذار و کدگشا را دارند. این لایه‌ها به عنوان پلی بین استخراج ویژگی‌ها (کدگذار) و بازسازی ویژگی‌ها (کدگشا) عمل می‌کنند. در این قسمت یک لایه کانولوشن دو بعدی استفاده می‌کنیم که `stride=1,kernel=3` باشد.

۶.۳.۵ کدگشا [۲]

بخش کدگشا در سمت راست معماری این شبکه واقع شده است. این بخش مسئول افزایش ابعاد نقشه‌های ویژگی و تولید خروجی با رزولوشن بالا می‌باشد. این قسمت دارای دو بخش اصلی است:

- `upconvx` : در این قسمت از `ConvTranspose2d` برای نمونه برداری استفاده می‌شود، که نقشه ویژگی خروجی را بزرگتر از نقشه ویژگی ورودی تولید می کند.
- `dconvx` : در این قسمت هر یک از بلاک ها مانند قسمت .کدگذار هستند(به ابعاد هر یک از بلوک ها دقت کنید)

۷.۳.۵ تابع forward

این تابع برای آموزش شبکه به کار می‌رود که داری سه بخش است:

- `Encoder` : در این قسمت باید عملیات downsampling انجام شود تا ویژگی‌ها استخراج شود، که از یک لایه max-pooling با `kernel_size=2, stride=2` استفاده می‌کنیم.
- `Bottleneck` : بلوکی که در قسمت قبل ساخته شده است را برای انجام این کار در نظر می‌گیریم.
- `Decoder` : در این قسمت عملیات Upsampling انجام می‌دهیم. در هر زیربخش از این قسمت یکسری skip connection وجود دارند که این اتصالات اطمینان می‌دهند که اطلاعات مکانی و جزئی که از دست نرود و به لایه‌های عمیق‌تر دسترسی داشته باشد، کمک به بهبود عملکرد شبکه و حفظ .اطلاعات معنی‌دار می‌کنند برای این کار نیاز است که بلوک conv قسمت کدگذار را به قسمت upconv متناظرش متصل کنید.

---

[۱] Bottleneck
[۲] Decoder