

گزارش جامع پروژه: طبقه‌بندی تصاویر با شبکه‌های عصبی در PyTorch



در این گزارش، ما سفری کامل از ابتدا تا انتهای یک پروژه یادگیری عمیق را مرور می‌کنیم. هدف اصلی این پروژه، ساخت، آموزش و ارزیابی مدل‌های شبکه عصبی برای طبقه‌بندی تصاویر از مجموعه داده معروف **CIFAR-10** بوده است. این گزارش تمام مراحل، از آماده‌سازی داده‌ها گرفته تا تعریف معماری‌های مختلف (شبکه تماماً متصل و شبکه کانولوشنی)، فرآیند آموزش و تحلیل نتایج را به زبانی ساده و قدم به قدم توضیح می‌دهد.

بخش ۱: آماده‌سازی اولیه و داده‌ها

هر پروژه یادگیری ماشین با یک پایه محکم شروع می‌شود: آماده‌سازی محیط و داده‌ها. در این بخش، ابزارهای لازم را فراهم کرده و داده‌های خام را به فرمتی مناسب برای استفاده در مدل‌هایمان تبدیل می‌کنیم.

۱.۱. وارد کردن کتابخانه‌ها و تنظیمات اولیه

✓ کتابخانه‌های اصلی: از کتابخانه `torch` (که PyTorch است) و زیرماژول‌های آن مانند `torch.nn` (برای ساخت شبکه‌های عصبی) و `torchvision` (برای کار با مجموعه داده‌های تصویری و تبدیل‌ها) استفاده می‌کنیم.

✓ انتخاب دستگاه :**(Device)**

```
'device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

این کد به طور هوشمند بررسی می‌کند که آیا GPU با قابلیت CUDA در دسترس است یا خیر. اگر باشد، محاسبات روی GPU انجام می‌شود (که برای آموزش شبکه‌های عصبی بسیار سریع‌تر است)، در غیر این صورت از CPU استفاده خواهد شد.

✓ تعریف کلاس‌ها :**(Labels)**

```
classes = ['plane', 'car', ..., 'truck']
```

مجموعه داده CIFAR-10 شامل ۱۰ کلاس مختلف از تصاویر است که نام آن‌ها را در این لیست برای استفاده‌های بعدی (مانند نمایش نتایج) ذخیره می‌کنیم.

۱.۲. پیش‌پردازش تصاویر (Transforms)

تصاویر خام نمی‌توانند مستقیماً وارد شبکه عصبی شوند. آن‌ها نیاز به یک سری تبدیل‌ها دارند تا برای مدل قابل فهم شوند:

✓ `(transforms.ToTensor):` این تبدیل دو کار مهم انجام می‌دهد:

1. تصاویر را (که معمولاً با فرمت PIL یا NumPy هستند) به تانسورهای PyTorch تبدیل می‌کند. تانسورها ساختارهای داده اصلی در PyTorch هستند.

2. مقادیر پیکسل‌ها را از بازه [0, 255] به بازه [0.0, 1.0] نرمال می‌کند.

✓ `(transforms.Normalize(..., ...))`: مقادیر پیکسل‌های هر کanal رنگی (قرمز، سبز، آبی) را با استفاده از میانگین و انحراف معیار مشخصی نرمال می‌کند. این کار توزیع داده‌ها را بهینه کرده و به پایداری و سرعت فرآیند آموزش کمک می‌کند. مقادیر استفاده شده ((0.491, 0.247, 0.0) و (0.0, 0.0, 0.491)) مقادیر استاندارد محاسبه شده برای مجموعه داده CIFAR-10 هستند.

ما دو مجموعه تبدیل تعریف کردیم: `transform_train` برای داده‌های آموزشی و `transform_test` برای داده‌های آزمایشی و اعتبارسنجی.

۱.۳. بارگذاری و تقسیم‌بندی مجموعه داده CIFAR-10

✓ **بارگذاری اولیه:**

```
(...)initial_trainset = torchvision.datasets.CIFAR10  
(...)testset = torchvision.datasets.CIFAR10
```

مجموعه داده آموزشی (۵۰,۰۰۰ تصویر) و آزمایشی (۱۰,۰۰۰ تصویر) CIFAR-10 را با اعمال تبدیل‌های تعریف شده، بارگذاری می‌کنیم. اگر داده‌ها از قبل دانلود نشده باشند، به طور خودکار دانلود می‌شوند.

✓ **تقسیم داده‌های آموزشی:**

```
trainset, valset = random_split(initial_trainset, [45000, 5000])
```

مجموعه آموزشی اولیه را به دو بخش تقسیم می‌کنیم: یک مجموعه آموزشی جدید با ۴۰,۰۰۰ تصویر (`trainset`) و یک مجموعه اعتبارسنجی با ۵,۰۰۰ تصویر (`valset`). مجموعه اعتبارسنجی برای ارزیابی عملکرد مدل در طول آموزش و تنظیم هایپرپارامترها استفاده می‌شود.

۱.۴. ایجاد دیتا لودرهای (DataLoaders)

```
trainloader = DataLoader(trainset, batch_size=512, ...)  
valloader = DataLoader(valset, batch_size=512, ...)  
testloader = DataLoader(testset, batch_size=512, ...)
```

برای هر یک از مجموعه‌های داده (آموزشی، اعتبارسنجی، آزمایشی)، یک `DataLoader` ایجاد می‌کنیم. `DataLoader` وظایف مهمی را بر عهده دارد:

- **دسته‌بندی (Batching):** داده‌ها را به دسته‌های کوچک (در اینجا با اندازه ۵۱۲) تقسیم می‌کند. آموزش با بجها به جای کل داده‌ها به یکباره، کارآمدتر است.
- **به هم زدن (Shuffling):** برای `trainloader`، گزینه `shuffle=True` داده‌ها را در هر اپوک به هم می‌زند تا از یادگیری الگوهای ناخواسته ناشی از ترتیب داده‌ها جلوگیری شود. برای `valloader` و `testloader` معمولاً نیازی به این کار نیست.
- **بارگذاری موازی (Parallel Loading):** با استفاده از `num_workers` می‌تواند داده‌ها را به صورت موازی بارگذاری کند که سرعت آموزش را افزایش می‌دهد.



بخش ۳: ابزارهای کمکی و بصری‌سازی اولیه

۱.۱. کلاس `UnNormalize`

از آنجایی که تصاویر ما نرمال‌سازی شده‌اند، برای نمایش صحیح آن‌ها به شکلی که برای انسان قابل درک باشد، نیاز به یک تابع یا کلاس برای معکوس کردن این نرمال‌سازی داریم. کلاس `UnNormalize` این کار را انجام می‌دهد:

/در متدهای `__init__`، میانگین و انحراف معیاری که برای نرمال‌سازی استفاده شده‌اند را دریافت می‌کند.

/در متدهای `__call__`، یک تansور تصویر نرمال‌شده را گرفته و با اعمال فرمول معکوس ($\text{تصویر_نرمال_شده} * \text{انحراف_معیار} + \text{میانگین}$) آن را به مقادیر اصلی‌تر برگرداند.

/یک نمونه از این کلاس با نام `norminv` با استفاده از میانگین و انحراف معیار CIFAR-10 ایجاد شد.

۱.۲. بصری‌سازی نمونه تصاویر

برای درک بهتر داده‌ها، ۰ تصویر تصادفی از هر کلاس در مجموعه داده آموزشی انتخاب و نمایش داده شد:

/ابتدا تصاویر بر اساس کلاس‌هایشان گروه‌بندی شدند.

/سپس از هر کلاس، ۰ تصویر به صورت تصادفی انتخاب شد.

/این تصاویر با استفاده از `norminv` به حالت غیرنرمال درآمدند.

در نهایت، با استفاده از کتابخانه `matplotlib.pyplot` تصاویر در یک گرید نمایش داده شدند که هر ستون به یک کلاس اختصاص داشت.

بخش ۳: تعریف مدل شبکه عصبی

در این پروژه، دو نوع مدل شبکه عصبی را بررسی کردیم: ابتدا یک شبکه تماماً متصل (Fully Connected Network - FCN) و سپس یک شبکه عصبی کانولوشنی (Convolutional Neural Network - CNN) که معمولاً برای وظایف بینایی کامپیوتر عملکرد بهتری دارد. هدف اصلی، طراحی مدلی با حدود ۳۳.۰ میلیون پارامتر قابل آموزش بود.

۱.۳. مدل اول: شبکه تماماً متصل (FCN) - (به طور خلاصه)

یک کلاس `nn.Module` تعریف شد که از `FullyConnectedNetwork` ارث بری می‌کرد. ساختار آن شامل:

✓ یک لایه `nn.Flatten()` برای صاف کردن تصویر ورودی.

✓ یک بلوک لایه پنهان (`self.hidden_layers`) شامل:

- یک `nn.Linear` برای نگاشت ورودی به فضای پنهان (با `h1_features = 10860` نورون).

- یک `nn.BatchNorm1d` برای نرمال‌سازی دسته‌ای.

- یک تابع فعال‌سازی `nn.ReLU()`.

- یک لایه `nn.Dropout` برای جلوگیری از بیش‌برازش.

✓ یک لایه خروجی `nn.Linear` (با نام `self.linear`) برای نگاشت فضای پنهان به ۱۰ کلاس خروجی.

✓ متد `forward` نیز برای تعریف نحوه عبور داده‌ها از این لایه‌ها پیاده‌سازی شد.

تعداد پارامترهای این مدل با استفاده از `torchsummary` بررسی شد تا در محدوده خواسته شده (حدود ۳۳.۰ میلیون) باشد.

۲.۳. مدل دوم (و نهایی): شبکه عصبی کانولوشنی (CNN)

برای بهبود عملکرد، یک مدل CNN با نام کلاس `CNN` طراحی و پیاده‌سازی شد. مدل‌های CNN با استفاده از لایه‌های کانولوشنی، ویژگی‌های محلی و سلسله مراتبی تصاویر را بهتر استخراج می‌کنند.

۲.۳.۳. مفاهیم کلیدی در CNN (برای تازه‌کاران)

• لایه کانولوشنی (`nn.Conv2d`) : با اعمال فیلترهایی روی تصویر، نقشه‌های ویژگی (feature maps) را استخراج می‌کند (مانند لبه‌ها، بافت‌ها و ...).

• تابع فعال‌سازی (`nn.ReLU`) : غیرخطی بودن را به شبکه اضافه می‌کند و به یادگیری الگوهای پیچیده‌تر کمک می‌کند.

• نرمال‌سازی دسته‌ای (`nn.BatchNorm1d` و `nn.BatchNorm2d`) : خروجی لایه‌ها را نرمال می‌کند، که به پایداری و سرعت آموزش کمک می‌کند.

• لایه ادغام (`Pooling Layer - nn.MaxPool2d`) : ابعاد نقشه‌های ویژگی را کاهش می‌دهد، تعداد پارامترها را کم می‌کند و مدل را نسبت به تغییرات جزئی در مکان ویژگی‌ها مقاوم‌تر می‌کند.

• لایه صاف‌کننده (`nn.Flatten`) : نقشه‌های ویژگی چندبعدی را به یک بردار تک بعدی تبدیل می‌کند تا بتوانند به ورودی لایه‌های تماماً متصل داده شوند.

• لایه تماماً متصل (`nn.Linear`) : مشابه آنچه در FCN دیدیم، برای طبقه‌بندی نهایی بر اساس ویژگی‌های استخراج شده استفاده می‌شود.

• لایه (`Dropout` (`nn.Dropout`) : از بیش‌برازش جلوگیری می‌کند.

CNN ساختار کلاس ۳.۵

این مدل به عنوان راهکار نهایی برای افزایش دقت طراحی شد. ساختار آن شامل دو بخش اصلی است:

بخش اول: پایه کانولوشنی (`self.conv_base`) این بخش برای استخراج ویژگی از تصاویر طراحی شده و شامل سه بلوک اصلی است. هر بلوک شامل دو لایه کانولوشنی (`Conv2d`)، دو لایه نرمال‌سازی (`BatchNorm2d`)، دو تابع فعال‌سازی (`ReLU`) و در نهایت یک لایه `MaxPool2d` برای کاهش ابعاد و یک لایه `Dropout` برای جلوگیری از بیش‌برازش است. با عمیق‌تر شدن شبکه (از بلوک ۱ به ۳)، تعداد فیلترها افزایش می‌یابد (۳۲ - > ۶۴ - > ۱۲۸) تا ویژگی‌های پیچیده‌تری استخراج شوند. خروجی این بخش، نقشه‌های ویژگی با ابعاد (128, 4, 4) است.

بخش دوم: پایه تماماً متصل (`Fully Connected Base`)

ابتدا، خروجی بخش کانولوشنی با `(``nn.Flatten`)` صاف شده و به یک بردار با ۲۰۴۸ ویژگی تبدیل می‌شود.

• سپس، `(``self.feature_fc_block`)` این بردار را به یک "فضای ویژگی" با ابعاد بزرگ (`self.feature_space_dim_N = 16200`) نگاشت می‌دهد. این بلوک نیز شامل لایه‌های `ReLU`، `Linear`، `BatchNorm1d`، `Dropout` و `(` است.

• در نهایت، `(``self.linear`)` (لایه خروجی) این فضای ویژگی ۱۶۲۰۰ بعدی را به ۱۰ کلاس نهایی نگاشت می‌دهد.

طراحی این بخش به گونه‌ای انجام شد که تعداد کل پارامترهای مدل در محدوده حدود ۳۳.۰ میلیون باقی بماند.

۳.۲.۳. خلاصه و نمونه‌سازی مدل CNN

✓ یک نمونه از کلاس `CNN` ساخته شد.

✓ ساختار آن با استفاده از `torchsummary` برای بررسی لایه‌ها و تعداد پارامترها نمایش داده شد.

✓ مدل با استفاده از `.` به دستگاه انتخاب شده (CPU یا GPU) منتقل شد.

بخش ۴: فرآیند آموزش مدل

پس از تعریف مدل، باید آن را آموزش دهیم. این فرآیند شامل تعریفتابع هزینه، بهینه‌ساز، و اجرای یک حلقه آموزشی برای چندین اپوک است.

۴.۱. تعریف تابع هزینه (Criterion) و بهینه‌ساز (Optimizer)

تابع هزینه: `(criterion = nn.CrossEntropyLoss()`

از `CrossEntropyLoss` استفاده شد که برای مسائل طبقه‌بندی چندکلاسه استاندارد است. این تابع هزینه به طور داخلی شامل Softmax نیز می‌شود.

بهینه‌ساز: `optimizer = torch.optim.Adam(model.parameters(), lr=0.001)`

بهینه‌ساز `Adam` به دلیل کارایی و نیاز کمتر به تنظیم دقیق هایپرپارامترها انتخاب شد. این بهینه‌ساز پارامترهای مدل را با نرخ یادگیری (`lr`) مشخص شده به روزرسانی می‌کند.

زمان‌بند نرخ یادگیری (Learning Rate Scheduler): (در کد نهایی کامنت شده بود)

بحث شد که استفاده از یک زمان‌بند مانند `torch.optim.lr_scheduler.StepLR` می‌تواند با کاهش تدریجی نرخ یادگیری در طول آموزش، به همگرایی بهتر کمک کند.

۴.۲. توابع آموزش و ارزیابی (`eval_epoch` و `train_epoch`)

دو تابع اصلی برای مدیریت حلقه آموزش پیاده‌سازی شد:

✓ این تابع مدل را برای یک اپوک کامل روی داده‌های آموزشی آموزش می‌دهد. مراحل اصلی آن شامل: قرار دادن مدل در حالت آموزش (`net.train()`)، صفر کردن گرادیان‌ها (`optimizer.zero_grad()`)، پاس رو به جلو (`net(inputs)`)، محاسبه هزینه (`loss.backward()`)، پاس رو به عقب (`criterion(loss)` برای محاسبه گرادیان‌ها، و به روزرسانی پارامترها (`optimizer.step()`)) است. در نهایت، میانگین هزینه و دقت برای آن اپوک را برمی‌گرداند.

(`eval_epoch(...)`): این تابع عملکرد مدل را روی داده‌های اعتبارسنجی یا آزمایشی برای یک اپوک ارزیابی می‌کند. این تابع مدل را در حالت ارزیابی (`net.eval()`) قرار می‌دهد و تمام محاسبات را داخل `with torch.no_grad():` انجام می‌دهد تا از محاسبه گرادیان جلوگیری شود. این تابع نیز میانگین هزینه و دقت را برمی‌گرداند.

۴.۳. حلقه اصلی آموزش

یک حلقه `for` برای ۶۰ اپوک اجرا شد. در هر اپوک:

- ✓ تابع `trainloader` با `train_epoch` فراخوانی شد تا مدل آموزش ببیند.
- ✓ تابع `valloader` با `eval_epoch` فراخوانی شد تا عملکرد مدل روی داده‌های اعتبارسنجی ارزیابی شود.
- ✓ هزینه و دقت آموزشی و اعتبارسنجی به دست آمده در دیکشنری `history` برای رسم نمودار ذخیره شدند.
- ✓ نتایج هر اپوک در خروجی چاپ شد تا پیشرفت آموزش قابل مشاهده باشد.
- ✓ پس از اتمام حلقه، وضعیت پارامترهای مدل آموزش دیده با `torch.save(model.state_dict(), "cnn.pth")` در یک فایل ذخیره شد تا بعداً بتوان از آن استفاده کرد.

بخش ۵: نتایج و تحلیل

۵.۱. بصری‌سازی نتایج آموزش

یک تابع به نام `plot_training_results` برای رسم نمودارها استفاده می‌کند:

- ✓ **نمودار دقت:** این نمودار، دقت آموزشی و دقت اعتبارسنجی را در طول اپوک‌ها نمایش می‌دهد. این نمودار برای تشخیص بیش‌برازش (زمانی که دقت آموزش بالا می‌رود اما دقت اعتبارسنجی ثابت می‌ماند یا کاهش می‌یابد) یا کم‌برازش بسیار مفید است.
- ✓ **نمودار هزینه:** این نمودار، هزینه آموزشی و هزینه اعتبارسنجی را نشان می‌دهد. در یک آموزش موفق، هر دو هزینه باید به طور کلی کاهش یابند.

۵.۲. ارزیابی نهايی داده‌های آزمایشي

در نهايیت، برای سنجش عملکرد واقعی مدل، آن را روی مجموعه داده آزمایشي (`testset`) که تاکنون در فرآيند آموزش دیده نشده بود، ارزیابی كردیم:

تابع `eval_epoch` با `testloader` فراخوانی شد.

میانگین هزینه و دقت نهایی مدل روی داده‌های آزمایشی چاپ شد.

این دقت، معیار اصلی برای قضاوت در مورد عملکرد نهایی مدل ما است.

نتیجه‌گیری و گام‌های بعدی

در این پژوهه، ما با موفقیت یک پایپلاین کامل یادگیری عمیق را از ابتدا تا انتها پیاده‌سازی کردیم. با داده‌های CIFAR-10 شروع کردیم، آن‌ها را پیش‌پردازش کردیم، دو نوع معماری (CNN و FCN) را با محدودیت‌های خاص طراحی کردیم، مدل CNN نهایی را آموزش دادیم، نتایج را بصری‌سازی کردیم و عملکرد نهایی آن را سنجیدیم.

این پژوهه یک پایه محکم برای درک مفاهیم کلیدی PyTorch و فرآیند ساخت مدل‌های بینایی کامپیوتر فراهم می‌کند. گام‌های بعدی برای بهبود بیشتر می‌تواند شامل موارد زیر باشد:

→ تنظیم دقیق‌تر هایپرپارامترها (نرخ یادگیری، اندازه بج، مقادیر Dropout و ...).

→ استفاده از تکنیک‌های افزایش داده (Data Augmentation) قوی‌تر.

→ آزمایش با معماری‌های CNN پیشرفته‌تر (مانند معماری‌های مبتنی بر ResNet).

→ استفاده از یادگیری انتقال (Transfer Learning) با استفاده از مدل‌های از پیش آموزش دیده.

تحليل دقیق و خط به خط معماری مدل‌ها

در این بخش، دو معماری شبکه‌ای که در طول پژوهه طراحی و پیاده‌سازی کردیم، یعنی شبکه تمام‌اً متصل (Fully Connected) و شبکه عصبی کانولوشنی (Convolutional Neural Network) را به صورت کاملاً دقیق و خط به خط بررسی می‌کنیم. هدف این است که درک عمیقی از نحوه کارکرد هر لایه و منطق پشت طراحی آن‌ها به دست آوریم، به گونه‌ای که برای یک فرد تازه‌کار نیز قابل فهم باشد.

۱. مدل اول: شبکه تماماً متصل (Fully Connected Network - FCN)

این مدل، یک نوع کلاسیک از شبکه‌های عصبی است که در آن هر واحد پردازشی (نورون) در یک لایه به تمام نورون‌های لایه بعدی متصل است. این مدل برای درک مفاهیم اولیه و پایه‌ای شبکه‌های عصبی بسیار مناسب و آموزنده است.

کلاس `FullyConnectedNetwork`

الف) تعریف کلی کلاس

در خط `:class FullyConnectedNetwork(nn.Module)` تعریف می‌کنیم که از کلاس `nn.Module` در PyTorch می‌کند. `nn.Module` کلاس پایه‌ای برای تمام مدل‌های شبکه عصبی در PyTorch است و امکانات زیادی مانند ردیابی پارامترها، انتقال مدل به GPU و ذخیره‌سازی را برای ما فراهم می‌کند.

ب) متد سازنده (`__init__`)

این متد زمانی اجرا می‌شود که یک نمونه (object) از کلاس ساخته می‌شود. وظیفه آن تعریف و مقداردهی اولیه لایه‌های شبکه است.

```
:def __init__(self, input_shape=(3, 32, 32), num_classes=10, dropout_p=0.5)
```

این تعریف استاندارد متد سازنده است. `self` به خود نمونه کلاس اشاره دارد. `input_shape` ابعاد داده ورودی را مشخص می‌کند (۳ کanal، ارتفاع ۳۲، عرض ۳۲)، `num_classes` تعداد کلاس‌های خروجی (۱۰) و `dropout_p` احتمال Dropout را تعیین می‌کند.

```
()__super(FullyConnectedNetwork, self).__init__
```

این خط** بسیار مهم**، سازنده کلاس والد (`nn.Module`) را فراخوانی می‌کند تا تنظیمات پایه‌ای PyTorch برای مدل ما فعال شود.

```
input_features = input_shape[0] * input_shape[1] * input_shape[2]
```

در این خط، ابعاد تصویر ورودی را در هم ضرب می‌کنیم تا تعداد کل ویژگی‌ها (`3072 = 32 * 32 * 3`) برای ورودی لایه خطی به دست آید.

```
h1_features = 10860
```

این عدد، تعداد نورون‌های لایه پنهان است که برای رسیدن به تعداد کل پارامترهای هدف پروژه (حدود ۳۳.۰ میلیون) انتخاب شده است.

```
()self.flatten = nn.Flatten
```

برای صاف کردن ورودی چندبعدی به یک بردار تک بعدی ساخته و ذخیره می‌شود.

```
(... )self.hidden_layers = nn.Sequential
```

برای ساخت یک بلوک از لایه‌ها استفاده می‌کنیم که به ترتیب اجرا می‌شوند:

```
nn.Linear(input_features, h1_features) –  
نگاشت می‌دهد.  
nn.BatchNorm1d(h1_features) –  
نرمال‌سازی دسته‌ای برای پایداری آموزش.
```

```
(( )nn.ReLU –  
تابع فعال‌سازی برای افزودن غیرخطی بودن.  
nn.Dropout(p=dropout_p) –  
لایه Dropout برای جلوگیری از بیش‌برازش.
```

```
self.linear = nn.Linear(h1_features, num_classes)
```

لایه خروجی نهایی که خروجی ۱۰۸۶۰ بعدی از بلوک پنهان را به ۱۰ نورون خروجی (یکی برای هر کلاس) نگاشت می‌دهد.

ج) متدهای پیش‌رو (forward)

این متدهای پیش‌رو داده‌ها از شبکه را تعریف می‌کند.

```
:def forward(self, x)
```

این متدهای پیش‌رو داده ورودی `x` (یک بج از نمونه‌ها) را می‌گیرد.

```
x = self.flatten(x)
```

ورودی `x` توسط لایه `flatten` صاف می‌شود.

```
x = self.hidden_layers(x)
```

بردار صاف شده از تمام لایه‌های موجود در `hidden_layers` عبور می‌کند.

```
x = self.linear(x)
```

خروجی بلوک پنهان به لایه خروجی نهایی داده می‌شود.

```
return x
```

در نهایت، امتیازات خام (logits) برای هر کلاس برگردانده می‌شوند.

۵. مدل دوم (نهایی پروژه): شبکه عصبی کانولوشنی (CNN)

برای بهبود عملکرد، یک مدل CNN طراحی شد. این مدل‌ها برای داده‌های تصویری بسیار مناسب‌تر هستند زیرا می‌توانند ویژگی‌های فضایی (spatial features) را به خوبی استخراج کنند.

کلاس `CNN`

الف) متد سازنده (`__init__`)

```
:def __init__(self, num_classes=10, dropout_p_conv=0.5, dropout_p_fc=0.5)
```

سازنده کلاس CNN. دو پارامتر جداگانه برای Dropout تعریف شده است: `dropout_p_conv` برای بخش کانولوشنی و `dropout_p_fc` برای بخش تماماً متصل.

(`self.conv_base`) بخش کانولوشنی (

این بخش، پایه کانولوشنی مدل است که وظیفه اصلی استخراج ویژگی را بر عهده دارد و خود شامل سه بلوک اصلی است:

هر بلوک (Block 1, 2, 3) :

- `(...).nn.Conv2d` : لایه کانولوشنی. با عمیق‌تر شدن شبکه، تعداد فیلترها (کانال‌های خروجی) افزایش می‌یابد (مثلاً ۳۲ - ۶۴ - ۱۲۸).

- در هر بلوک، دو لایه از این نوع پشت سر هم قرار گرفته‌اند.

- `(...).nn.BatchNorm2d` : لایه نرمال‌سازی دسته‌ای برای داده‌های تصویری.

`(nn.ReLU)` : تابع فعال‌سازی.

`(nn.MaxPool2d)` : لایه ادغام که ابعاد فضای نقشه‌های ویژگی را نصف می‌کند.

`nn.Dropout(dropout_p_conv)` : لایه Dropout که بعد از هر بلوک برای جلوگیری از بیش‌برازش اعمال شده است.

خروجی نهایی `self.conv_base` : پس از عبور یک تصویر از این سه بلوک، خروجی یک نقشه ویژگی با ابعاد `(128, 4, 4)` خواهد بود.

بخش تماماً متصل (Fully Connected Part)

`self._conv_output_features = 128 * 4 * 4 # 2048`

تعداد کل ویژگی‌ها پس از صاف کردن خروجی بخش کانولوشنی محاسبه می‌شود.

`self.feature_space_dim_N = 16200`

ابعاد "فضای ویژگی" که برای حفظ تعداد کل پارامترها در محدوده هدف پروژه تنظیم شده است.

`(...)self.feature_fc_block = nn.Sequential`

این بلوک، بردار ویژگی صاف شده را به فضای ویژگی `16200` بعدی نگاشت می‌دهد.

`(...)self.linear = nn.Linear`

لایه خروجی نهایی که بردار فضای ویژگی را به ${}^{10}\text{}`$ کلاس نهایی نگاشت می‌دهد.

ج) متد پیش‌رو (`forward`)

این متد مسیر عبور داده‌ها از لایه‌های مختلف مدل CNN را تعریف می‌کند.

`:def forward(self, x: torch.Tensor) -> torch.Tensor`

ورودی `x` را می‌گیرد.

`x = self.conv_base(x)`

ورودی از بلوک‌های کانولوشنی عبور می‌کند.

`x = self.flatten(x)`

نقشه‌های ویژگی حاصل، صاف می‌شوند.

```
feature_space_output = self.feature_fc_block(x)
```

بردار صاف شده به فضای ویژگی نگاشت داده می‌شود.

```
out_logits = self.linear(feature_space_output)
```

بردار فضای ویژگی به لایه خروجی نهایی داده می‌شود.

```
return out_logits
```

امتیازات نهایی (logits) برگردانده می‌شوند.