به نام خدا

پروژه چهارم درس یادگیری عمیق
آشنایی با شبکه کانولوشنی و یادگیری انتقالی

استاد درس: دکتر صفابخش

نگارش: زهرا اخلاقی

شماره دانشجویی: ۴۰۱۱۳۱۰۶۴

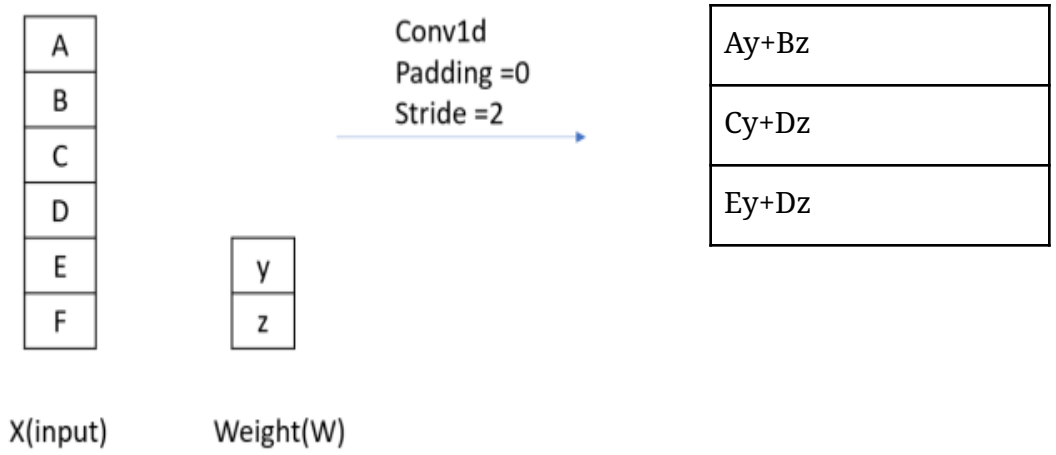
پاییز ۱۴۰۲

فهرست مطالب

بخش‌اول 2
۱- الف 2
۲- الف 2
۲- ب 2
۲- ج 3
۳- الف 4
۳- ب 5
بخش‌دوم 7
۱ 7
۲ 9
بخش‌سوم 12
۱ 12
۲ 13

بخش‌اول

۱- الف



۲- الف

نقش لایه اول در شبکه‌های عصبی کانولوشنی، استخراج ویژگی های سلسله مراتبی از داده های ورودی، حفظ اطلاعات مکانی و آماده سازی داده ها برای پردازش بیشتر در لایه های بعدی شبکه است. لایه‌اول عملیات کانولوشن را روی داده های ورودی اعمال می کند، در این عمل یک فیلتر کوچک (ماتریسی از وزن ها) روی ورودی می لغزد و در هر موقعیت یک حاصل ضرب نقطه ای را محاسبه می کند. نتیجه یک نقشه ویژگی است که اطلاعاتی در مورد حضور و آرایش فضایی الگوهای خاص در ورودی ثبت می‌کند. لایه اول معمولاً بر تشخیص عناصر بصری اساسی مانند لبه ها، خطوط و تضاد رنگ تمرکز می کند. این ویژگی ها بلوک های پایه برای ویژگی های پیچیده تر استخراج شده توسط لایه های بعدی را تشکیل می دهند.

۲- ب

اندازه هسته یا اندازه فیلتر در لایه های اولیه یک شبکه عصبی کانولوشنال (CNN) نقش مهمی در استخراج ویژگی و عملکرد کلی شبکه ایفا می کند. هسته های بزرگتر (5x5 یا بیشتر): روابط فضایی گسترده تر و الگوهای پیچیده را ثبت کنید. برای استخراج ویژگی های کلی مانند اشکال شی یا بافت مفید است، اما از نظر محاسباتی گران و مستعد overfitting در

مجموعه داده های کوچک می‌باشد. هسته های کوچکتر (3x3 یا 1x1) جزئیاتی مانند لبه ها، گوشه ها و اشکال اصلی را استخراج کنید. از نظر محاسباتی کارآمدتر و کمتر مستعد overfitting هستند اما زمینه و روابط بزرگتر را از دست می‌دهد.

هسته‌های بزرگ‌تر، قسمت بیشتری از تصویر ورودی را پوشش می‌دهند. هسته‌های کوچک‌تر نقشه‌های ویژگی کوچک‌تری تولید می‌کنند که برای دستیابی به همان سطح پیچیدگی به لایه‌های بیشتری نیاز دارند. هسته های بزرگتر به حافظه و قدرت پردازش بیشتری نیاز دارند و در کارهای پیچیده به هسته های بزرگتری برای ثبت جزئیات پیچیده نیاز است.

لایه های اولیه معمولاً از هسته های کوچکتر (3×3) برای استخراج ویژگی های اساسی بدون از دست دادن اطلاعات مکانی استفاده می شود. اما در لایه‌های میانی از ترکیبی از هسته‌های کوچک و بزرگ‌تر برای ایجاد ویژگی‌های اساسی و استخراج قطعات شی خاص استفاده شود. لایه‌های بعدی می‌تواند از هسته‌های بزرگ‌تر (۵×۵ یا حتی ۷×۷) برای ثبت روابط جهانی و زمینه بین ویژگی‌ها استفاده کند. مجموعه داده های بزرگتر می توانند هسته های بزرگتر را پشتیبانی کنند، در حالی که مجموعه داده های کوچکتر به هسته های کوچکتری برای جلوگیری از overfitting نیاز دارند.

۲- ج

Spatial pooling یک عملیات حیاتی در CNN است که ابعاد نقشه های ویژگی استخراج شده توسط لایه های کانولوشن را کاهش می دهد و دارای اهداف زیر میباشد:

۱- با کوچک کردن اندازه نقشه‌های ویژگی، ادغام تعداد پارامترها و محاسبات مورد نیاز در لایه‌های بعدی را کاهش می‌دهد و شبکه را سریع‌تر و سبک‌تر می‌کند.

۲-حساسیت شبکه را نسبت به تغییرات کوچک در ورودی کاهش می‌دهد، و استحکام آن را در برابر نویز و اعوجاج بهبود می‌بخشد.

۳- به شبکه اجازه می‌دهد تا با خلاصه کردن اطلاعات مناطق کوچک‌تر، روی ویژگی‌های عمومی‌تر و ثابت‌تر تمرکز کند.

چندین مکانیسم متداول برای pooling وجود دارد:

Max Pooling: حداکثر مقدار را از یک منطقه تعریف شده (به عنوان مثال، 2x2 pool) در نقشه ویژگی انتخاب می کند. در برجسته کردن ویژگی های غالب و مکان آنها موثر است. نسبت به روش های دیگر حساسیت کمتری به نویز دارد. اغلب برای کارهایی مانند تشخیص و تشخیص اشیا ترجیح داده می شود، جایی که شناسایی قوی ترین ویژگی ها و مکان آنها بسیار مهم است، استفاده می‌شود.

Average Pooling: مقدار متوسط را در یک منطقه تعریف شده در نقشه ویژگی محاسبه می کند. اطلاعات مربوط به همه ویژگی‌های موجود در منطقه، نه فقط قوی‌ترین‌ها را ضبط می‌کند. می تواند از نظر محاسباتی کارآمدتر از Max Pooling باشد. این pooling میتواند برای کارهایی مانند طبقه بندی تصویر موثر باشد، جایی که گرفتن اطلاعات در مورد همه ویژگی های یک منطقه مهم است، استفاده شود.

Sum Pooling: مجموع همه مقادیر را در یک منطقه تعریف شده در نقشه ویژگی محاسبه می کند. مشابه Average Pooling است، اما می تواند تاثیر مقادیر بزرگتر را تقویت کند. کمتر از MAX یا AVG در CNN های استاندارد استفاده می شود، اما می تواند در سناریوهای خاصی که تاکید بر مقادیر بزرگتر سودمند است، مفید باشد.

انتخاب Min Pooling در یک منطقه تعریف شده به ندرت در CNN های استاندارد استفاده می شود. ممکن است در کاربردهای خاص در نظر گرفته شود که در آن شناسایی یا سرکوب نویز پس‌زمینه یا ویژگی‌های ضعیف ضروری است. با این حال، کاربرد کلی آن در مقایسه با سایر مکانیسم‌های pooling محدود است. توجیه استفاده از min pooling به شرح زیر است:

۱- برخلاف max pooling که بر قوی‌ترین فعال‌سازی‌ها تمرکز دارد، min pooling بر ضعیف‌ترین فعال‌سازی‌ها در یک منطقه تعریف‌شده تأکید می‌کند و می تواند برای کارهایی مفید باشد که شناسایی مناطق تاریک، بافت های ظریف یا ویژگی های کم شدت بسیار مهم است.

۲- به‌طور مؤثر نویز یا اطلاعات پس‌زمینه را که ممکن است در نقشه‌های ویژگی وجود داشته باشد، سرکوب کند. با تمرکز بر ضعیف‌ترین فعال‌سازی‌ها، تأثیر نوسانات تصادفی یا جزئیات نامربوط را کاهش می‌دهد و به طور بالقوه منجر به نمایش تمیزتر برای لایه‌های بعدی می‌شود.

کاربردها:

Min pooling می تواند در برنامه های کاربردی خاص مانند: تجزیه و تحلیل تصویر پزشکی(شناسایی ناهنجاری های ظریف در مناطق با شدت کم در اسکن هایی مانند اشعه ایکس)، ستاره شناسی (استخراج سیگنال های ضعیف از اجرام آسمانی در پس زمینه تاریک فضا) و رانندگی خودمختار (تشخیص علائم یا موانع ظریف جاده با کنتراست کم) استفاده می‌شود.

چالش ها و ملاحظات:

- مرکز صرف بر روی ضعیف‌ترین فعال‌سازی‌ها می‌تواند منجر به از دست دادن اطلاعات ارزشمند موجود در فعال‌سازی‌های قوی‌تر شود. متعادل کردن این مبادله مستلزم بررسی دقیق کار و مجموعه داده است.
- انتخاب اندازه هسته و گام مناسب برای برای دستیابی به اثرات مطلوب آن بسیار مهم است. هسته های بیش از حد بزرگ ممکن است تمام اطلاعات معنی دار را حذف کنند، در حالی که هسته های کوچک ممکن است زمینه کافی را دریافت نکنند.

۳- الف

شبکه‌های عصبی عمیق از ناپدید شدن گرادیان رنج می‌برند، زیرا از گرادیان که برای به‌روزرسانی وزن‌ها در طول تمرین استفاده می‌کنند و این مقدار با انتشار در شبکه بسیار کوچک می‌شوند. این اساساً تأثیرگذاری لایه‌های قبلی بر یادگیری لایه‌های بعدی را دشوار می‌کند. اتصالات باقیمانده برای رفع این مشکل پیشنهاد شده‌اند. آنها اتصالات پرش ایجاد می کنند

که مستقیماً ورودی یک لایه را به خروجی آن اضافه می کنند و چندین لایه میانی را دور می زنند. این به اطلاعات لایه‌های قبلی اجازه می‌دهد تا مستقیماً به لایه‌های بعدی سرازیر شوند و مشکل ناپدید شدن گرادیان را کاهش دهد.
ایجاد اتصالات باقیمانده در شبکه‌های عصبی، مانند شبکه‌های باقیمانده (ResNets)، به موضوع ناپدید شدن گرادیان می‌پردازد. شبکه‌های عمیق سنتی از ناپدید شدن گرادیان رنج می‌برند، که آموزش مدل‌های بسیار عمیق را دشوار می‌کند. اتصالات باقیمانده راهی برای کاهش این مشکل با اجازه دادن به گرادیان برای جریان آسان تر از طریق شبکه در طول آموزش فراهم می کند.
در شبکه‌های عمیق سنتی، با عمیق‌تر شدن شبکه، گرادیان‌ها بسیار کوچک شده زیرا در لایه‌ها به عقب انتشار می‌یابند، که منجر به مشکل ناپدید شدن گرادیان می‌شود. این امر آموزش موثر شبکه های عمیق را چالش برانگیز می کند. از سوی دیگر، اتصالات باقیمانده، به گرادیان اجازه می دهد تا با افزودن ورودی یک لایه به خروجی، لایه های خاصی را دور بزند. این به این معنی است که گرادیان مسیر کوتاه تری برای حرکت در حین انتشار پس زمینه دارد که به کاهش مشکل گرادیان ناپدید شدن کمک می کند.
در نتیجه، اتصالات باقیمانده آموزش شبکه‌های عمیق‌تر را در مقایسه با معماری‌های سنتی امکان‌پذیر می‌سازد. بنابراین، معرفی اتصالات باقیمانده در شبکه‌های عصبی، همانطور که در ResNets و سایر معماری‌های مرتبط مشاهده می‌شود، مستقیماً با پرداختن به موضوع ناپدید شدن گرادیان مرتبط است، و آموزش مدل‌های بسیار عمیق را به طور مؤثرتری امکان‌پذیر می‌سازد.

۳- ب

طول و تعداد اتصالات باقیمانده تاثیر قابل توجهی بر انتشار ویژگی در شبکه های عمیق دارد. در شبکه‌های کانولوشن سنتی، خروجی هر لایه به لایه بعدی آن متصل می‌شود، که می‌تواند با عمیق‌تر شدن شبکه به مشکل محو شدن گرادیان منجر شود.
طول و تعداد اتصالات باقیمانده تأثیر پیچیده و ظریفی بر انتشار ویژگی در شبکه‌های عصبی دارد:

اثرات مثبت

۱- یک اتصال طولانی باقی مانده می تواند ویژگی‌هایی که فاصله زیادی از یکدیگر دارند را مستقیماً به هم پیوند دهد و به شبکه اجازه می دهد تا همبستگی های پیچیده بین آنها را یاد بگیرد. ۲- با مسیرهای مستقیم بیشتر، گرادیان ها می توانند به طور موثرتری به عقب انتشار پیدا کنند، به خصوص در شبکه های بسیار عمیق، که منجر به یادگیری بهتر می شود.۳-هر اتصال اطلاعات لایه های قبلی را تزریق می کند و ویژگی های موجود برای لایه های بعدی را غنی می کند. ۴- اتصالات متعدد به یک لایه باعث می شود شبکه کمتر مستعد اختلال در هر مسیر باشد. این امر ثبات و تعمیم پذیری را بهبود می بخشد.

اثرات منفی

۱- پردازش اطلاعات در مسیرهای طولانی تر به منابع بیشتر، افزایش زمان آموزش و استفاده از حافظه نیاز دارد.

۲- جریان بیش از حد اطلاعات می تواند باعث شود شبکه به جای یادگیری الگوهای تعمیم پذیر، داده های آموزشی را به خاطر بسپارد. این به عملکرد در داده های دیده نشده آسیب می زند.اتصالات بیشتر به معنای عملیات بیشتر است و نیازهای محاسباتی را تشدید می کند.

به طور کلی، تعداد زیاد میانبرهای طولانی میتواند منجر به Overfitting شود. برای همین باید در نظر داشت که بین انتشار ویژگی موثر و پارامترسازی باید یک تعادلی وجود داشته باشد.

دو نمونه از موارد استفاده:

ResNets با معرفی اتصالات پرش که تبدیل‌های غیرخطی را با یک تابع هویت دور می‌زند، به این موضوع پرداخت و به گرادیان اجازه می‌دهد مستقیماً از طریق تابع هویت از لایه‌های بعدی به لایه‌های قبلی جریان یابد.

DenseNets با معرفی اتصالات مستقیم از هر لایه به تمام لایه های بعدی، این مفهوم را بیشتر می کند، در نتیجه جریان اطلاعات بین لایه ها را بهبود می بخشد. این الگوی اتصال متراکم به هر لایه اجازه می‌دهد تا ورودی‌های اضافی را از تمام لایه‌های قبلی به دست آورد و نقشه‌های ویژگی خود را به تمام لایه‌های بعدی منتقل کند، که در نتیجه اتصالات $L(L+1)/2$ در یک شبکه لایه L ایجاد می‌شود. این الگوی اتصال متراکم استفاده مجدد از ویژگی ها را تشویق می کند و با اطمینان از اینکه هر لایه به نقشه های ویژگی از همه لایه های قبلی دسترسی دارد، مشکل کاهش گرادیان را کاهش می دهد، بنابراین انتشار ویژگی در سراسر شبکه را تسهیل می کند.

در RoR (Residual Networks of Residual Networks) افزودن اتصالات باقیمانده بیشتر می تواند توانایی بهینه سازی شبکه را بهبود بخشد و منجر به عملکرد بهتر در مجموعه داده های مختلف شود. با این حال، اضافه کردن تعداد زیاد اتصالات باقیمانده نیز می تواند منجر به بیش از حد برازش شود، زیرا شبکه بسیار پیچیده می شود و به جای یادگیری ویژگی های قابل تعمیم، شروع به به خاطر سپردن داده های آموزشی می کند. علاوه بر این، طول اتصالات باقیمانده نیز می تواند بر انتشار ویژگی تأثیر بگذارد. اتصالات باقیمانده طولانی‌تر می‌توانند به انتشار مؤثرتر ویژگی‌ها از طریق شبکه کمک کنند، زیرا مسیر مستقیم‌تری را برای جریان گرادیان در طول انتشار پس‌پخش فراهم می‌کنند.

بخش‌دوم

این کد یک سری اعمال پیش پردازش روی تصاویر انجام می دهد تا برای ورودی مدل آماده شوند.

```
composed_train = transforms.Compose([transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)), # Resize the image in a 32X32 shape
                                     transforms.RandomRotation(20), # Randomly rotate some images by 20 degrees
                                     transforms.RandomHorizontalFlip(0.1), # Randomly horizontal flip the images
                                     transforms.ColorJitter(brightness = 0.1, # Randomly adjust color jitter of the images
                                                            contrast = 0.1,
                                                            saturation = 0.1),
                                     transforms.RandomAdjustSharpness(sharpness_factor = 2,
                                                                      p = 0.1), # Randomly adjust sharpness
                                     transforms.ToTensor(),   # Converting image to tensor
                                     transforms.Normalize(mean, std), # Normalizing with standard mean and standard deviation
                                     transforms.RandomErasing(p=0.75,scale=(0.02, 0.1),value=1.0, inplace=False)])


composed_test = transforms.Compose([transforms.Resize((IMAGE_SIZE,IMAGE_SIZE)),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean, std)])
```

دیتاست Cifar10 دارای دو قسمت train , test میباشد و 0.2 از داده‌های train برای validation انتخاب شده‌اند.

تابع train_model برای آموزش مدل می‌باشد در این تابع داده آموزش، مدل و optimizer، validation و تعداد ایپاک را به عنوان ورودی دارد.

۱

شبکه کانولوشنی طراحی شده برای این قسمت به صورت زیر می‌باشد، این مدل با استخراج ویژگی‌های از طریق لایه‌های کانولوشنال و تبدیل آن‌ها به لایه‌های کاملا متصل، تصویر را طبقه‌بندی می‌کند.

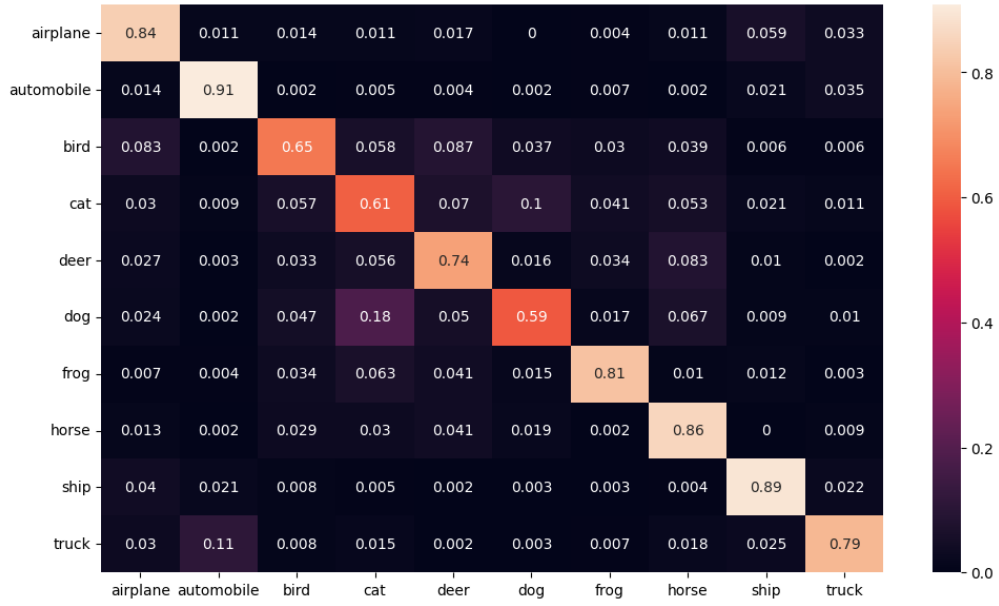
```
class CNN(nn.Module):
   def __init__(self, out_1 = 32, out_2 = 64, out_3 = 128, number_of_classes = 10, p = 0):
        super(CNN, self).__init__()
        self.cnn1 = nn.Conv2d(in_channels = 3, out_channels = out_1, kernel_size = 5, padding =
2)
        self.maxpool1 = nn.MaxPool2d(kernel_size = 2)

           self.cnn2 = nn.Conv2d(in_channels = out_1, out_channels = out_2, kernel_size = 5,
padding = 2)
        self.maxpool2 = nn.MaxPool2d(kernel_size = 2)

           self.cnn3 = nn.Conv2d(in_channels = out_2, out_channels = out_3, kernel_size = 5,
padding = 2)
        self.maxpool3 = nn.MaxPool2d(kernel_size = 2)

        # Hidden layer 1
        self.fc1 = nn.Linear(out_3 * 4 * 4, 1000)
         # 8x8 will change to 4x4 as we added a convolution & max pool layer refer calculation
comment above
        self.drop = nn.Dropout(p=p)
```

```
        # Hidden layer 2
        self.fc2 = nn.Linear(1000, 1000)

        # Final layer
        self.fc3 = nn.Linear(1000, 10)

    # Predictiona
    def forward(self, x):

        x = self.cnn1(x)
        x = torch.relu(x)
        x = self.maxpool1(x)

        x = self.cnn2(x)
        x = torch.relu(x)
        x = self.maxpool2(x)

        x = self.cnn3(x)
        x = torch.relu(x)
        x = self.maxpool3(x)

        x = x.view(x.size(0), -1)
        x = self.fc1(x)

        x = F.relu(self.drop(x))
        x = self.fc2(x)

        x = F.relu(self.drop(x))
        x = self.fc3(x)

        return(x)
```

این قطعه کد یک مدل شبکه عصبی کانولوشنال (CNN) برای طبقه‌بندی تصاویر است. این مدل سه لایه کانولوشنال دارد که هر کدام با اعمال فیلترهای 5x5 ویژگی‌های مهمی از تصویر استخراج می‌کنند. سپس، از لایه‌های Max Pooling برای کاهش ابعاد و جلوگیری از overfitting استفاده می‌کند. سپس، ویژگی‌های استخراج شده به لایه‌های کاملا متصل تبدیل می‌شوند. لایه اول 1000 نرون دارد و با ReLU فعال می‌شود. پس از آن، یک لایه Dropout احتمالاً (با احتمال p) برخی از نرون‌ها را خاموش می‌کند تا از overfitting جلوگیری کند. این مرحله با یک لایه کاملا متصل دیگر با 1000 نرون و فعال‌سازی ReLU تکرار می‌شود. در نهایت، لایه خروجی 10 نرون دارد که هر کدام احتمال تعلق تصویر به یک طبقه خاص را نشان می‌دهند.

دقت و ماتریس درهم‌ریختگی برای داده تست به صورت زیر میباشد:

```
confusion_matrix:
[[840  11  14  11  17   0   4  11  59  33]
 [ 14 908   2   5   4   2   7   2  21  35]
 [ 83   2 652  58  87  37  30  39   6   6]
 [ 30   9  57 606  70 102  41  53  21  11]
 [ 27   3  33  56 736  16  34  83  10   2]
 [ 24   2  47 183  50 591  17  67   9  10]
 [  7   4  34  63  41  15 811  10  12   3]
 [ 13   2  29  30  41  19   2 855   0   9]
 [ 40  21   8   5   2   3   3   4 892  22]
 [ 30 107   8  15   2   3   7  18  25 785]]
accuracy:0.7676
```



۲

اتصالات باقیمانده، به شبکه اجازه می دهند تا لایه های خاصی را دور بزند، که می تواند به جلوگیری از ناپدید شدن گرادیان کمک کند. برای افزودن اتصالات باقی‌مانده به شبکه کارهای زیر انجام شده است:

```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=5, stride=1, padding=2)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=5, stride=1, padding=2)

    def forward(self, x):
        residual = self.conv1(x)
        residual = self.relu(residual)
        residual = self.conv2(residual)

        return x + residual
```

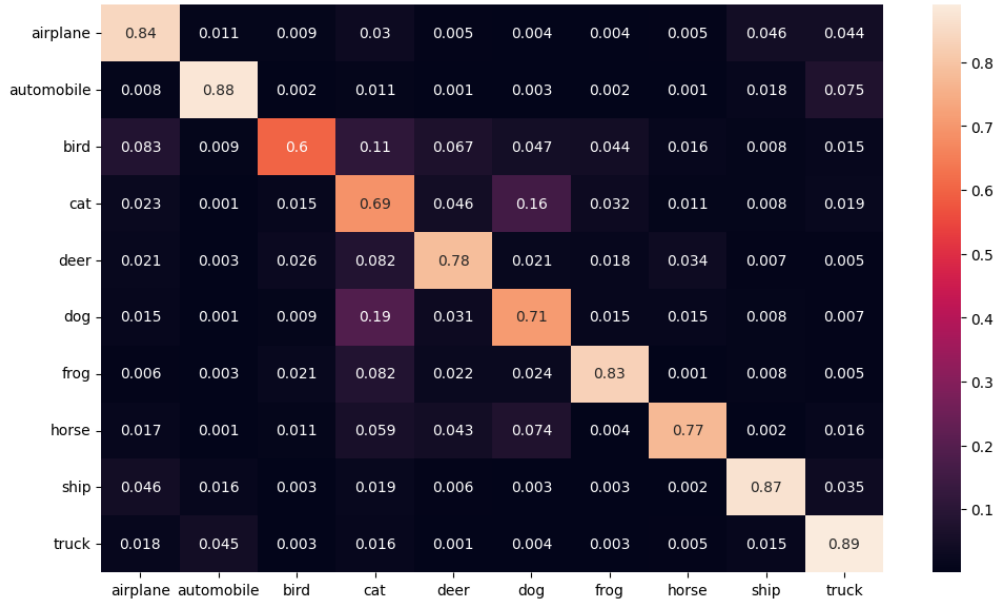
کلاس بالا از دو جزء تشکیل شده است: تابع باقیمانده و اتصال میانبر. تابع باقیمانده معمولاً از یک سری لایه های کانولوشنال، توابع فعال سازی ReLU و لایه های نرمال سازی دسته ای تشکیل شده است. اتصال میانبر به سادگی ورودی را مستقیماً به خروجی تابع باقیمانده منتقل می کند.

```
class ResidualCNN(nn.Module):
   def __init__(self, out_1 = 32, out_2 = 64, out_3 = 128, number_of_classes = 10, p = 0):
        super(ResidualCNN, self).__init__()
        self.cnn1 = nn.Conv2d(in_channels = 3, out_channels = out_1, kernel_size = 5, padding =
2)
        self.residual1 = ResidualBlock(out_1, out_1)
        self.maxpool1 = nn.MaxPool2d(kernel_size = 2)

           self.cnn2 = nn.Conv2d(in_channels = out_1, out_channels = out_2, kernel_size = 5,
padding = 2)
        self.residual2 = ResidualBlock(out_2, out_2)
        self.maxpool2 = nn.MaxPool2d(kernel_size = 2)

           self.cnn3 = nn.Conv2d(in_channels = out_2, out_channels = out_3, kernel_size = 5,
padding = 2)
        self.residual3 = ResidualBlock(out_3, out_3)
        self.maxpool3 = nn.MaxPool2d(kernel_size = 2)

        # Hidden layer 1
        self.fc1 = nn.Linear(out_3 * 4 * 4, 1000)
         # 8x8 will change to 4x4 as we added a convolution & max pool layer refer calculation
comment above
        self.drop = nn.Dropout(p=p)

        # Hidden layer 2
        self.fc2 = nn.Linear(1000, 1000)

        # Final layer
        self.fc3 = nn.Linear(1000, 10)
```

در forward CNN، خروجی تابع باقیمانده از طریق یک عنصر اضافه به ورودی اضافه می شود. این به طور موثر لایه های خاصی را "پرش" می کند و به شبکه اجازه می دهد تا نقشه برداری باقی مانده بین ورودی و خروجی را بیاموزد. این CNN دارای سه بلوک باقیمانده با 32 و 64 و 128 کانال است. اتصالات باقیمانده به شبکه اجازه می دهد تا نقشه پیچیده تری بین ورودی و خروجی را بیاموزد که می تواند عملکرد آن را در کار در دست بهبود بخشد. اتصالات باقیمانده ابزار قدرتمندی برای ساخت CNN های عمیق تر و قدرتمندتر هستند. با اجازه دادن به شبکه برای دور زدن لایه های خاص، آنها می توانند به جلوگیری از ناپدید شدن گرادیان کمک کنند، که می تواند یک مشکل بزرگ برای شبکه های عمیق باشد. در نتیجه، اتصالات باقیمانده به یک جزء ضروری از معماری مدرن CNN تبدیل شده است.

اتصالات باقی‌مانده می‌توانند سرعت آموزش و همگرایی مدل را بهبود بخشند. این امر به این دلیل است که اتصالات باقی‌مانده از ناپدید شدن گرادیان‌ها جلوگیری می‌کنند. زیرا ناپدید شدن گرادیان‌ها می‌تواند به آموزش نامنظم یا عدم رسیدن به همگرایی منجر شود. دقت و ماتریس درهم‌ریختگی برای داده تست به صورت زیر میباشد:

```
confusion_matrix:
[[842  11   9  30   5   4   4   5  46  44]
 [  8 879   2  11   1   3   2   1  18  75]
 [ 83   9 601 110  67  47  44  16   8  15]
 [ 23   1  15 689  46 156  32  11   8  19]
 [ 21   3  26  82 783  21  18  34   7   5]
 [ 15   1   9 190  31 709  15  15   8   7]
 [  6   3  21  82  22  24 828   1   8   5]
 [ 17   1  11  59  43  74   4 773   2  16]
 [ 46  16   3  19   6   3   3   2 867  35]
 [ 18  45   3  16   1   4   3   5  15 890]]
accuracy:0.7861
```



بخش سوم

در این بخش همچون بخش قبل در ابتدا پیش پردازشی روی تصاویر ورودی اعمال میشود. توزیع تصاویر ورودی به صورت زیر می‌باشد:

Data shapes (train/test):

(5000, 3, 96, 96)

(8000, 3, 96, 96)

Data value range:

(0, 255)

Data categories:

['airplane', 'bird', 'car', 'cat', 'deer', 'dog', 'horse', 'monkey', 'ship', 'truck']

۱

در این قسمت، ابتدا شبکه اول آموزش داده شده در بخش قبل بارگذاری می‌شود و وزن‌های آن ثابت می‌شوند، سپس یک شبکه پرسپترون با لایه مخفی دارای ۵۱۲ نورون به آن اضافه میشود، نتیجه روی داده تست به صورت زیر می‌باشد:

```
confusion_matrix:
[[538  47  23  10   6   3   4   1  79  89]
 [ 38 454  12 114  41  39  16  70  10   6]
 [ 11  14 626  13   0   8   9  12  18  89]
 [  9  43   6 445  81  68  49  74   7  18]
 [  9  35   2  90 531  28  77  20   3   5]
 [  6  55   7 144  57 281 142  98   6   4]
 [  2  19   4  51  31  59 569  47   2  16]
 [  4  90   8 190  39 104  77 275   1  12]
 [ 55  13  24  17   6   4   1   5 602  73]
 [ 33   5  87  27   2   6  30  19  38 553]]
 accuracy:0.60925
```

مشابه قسمت قبل این فرآیند برای شبکه کانولوشنی با اتصالات باقی‌مانده اجرا می‌شود که خروجی آن به صورت زیر است:

```
confusion_matrix:
[[590  32  12   4   8   5   5   4 102  38]
 [ 42 455  14  62  31  42  15 119  15   5]
 [  7  12 648   5   1  12   6   7  25  77]
 [  7  45   5 327  88 142  37 125  14  10]
 [  6  32   1  48 580  41  42  41   5   4]
 [  2  38   2  76  62 361 131 110   9   9]
 [  6  12   3  15  31  99 563  54   5  12]
 [  4  97   6  99  65 128  61 329   0  11]
 [ 52  16  15   8   3   3   4   9 655  35]
 [ 32  17  77   6   2   9  22  21  84 530]]
 accuracy:0.62975
```

۲

در این بخش همچون بخش قبل در ابتدا پیش پردازشی روی تصاویر ورودی اعمال میشود.

```
# Define STL10 data transformations
transform = transforms.Compose([
   transforms.Resize(299),  # InceptionV3 expects input size to be (299, 299)
   transforms.ToTensor(),
])

# Load STL10 dataset
stl10_train_dataset    =    datasets.STL10(root='/content/drive/MyDrive/stl10',    split='train',
download=True, transform=transform)
stl10_train_loader = DataLoader(stl10_train_dataset, batch_size=128, shuffle=True)

stl10_test_dataset         =     torchvision.datasets.STL10(root='/content/drive/MyDrive/stl10',
download=True, split='test',  transform=transform)
stl10_test_loader = DataLoader(stl10_test_dataset, batch_size=128)
```

مدل inception ابتدا load شده و سپس وزن های آن ثابت میشود و یک شبکه MLP به آن اضافه می‌شود، از CrossEntropy و بهینه ساز Adam تشکیل شده است.نتیجه روی داده آموزش پس از آموزش دو ایپاک به صورت زیر میباشد:

```
Epoch [1/2], Loss: 0.8522
Epoch [2/2], Loss: 0.4098
```

نتیجه روی داده تست به صورت زیر می‌باشد:

```
confusion_matrix:
[[779   1   1   0   0   1   1   0  17   0]
 [  5 777   0   4   0   8   1   5   0   0]
 [  1   0 775   1   0   0   2   0   1  20]
 [  0   8   0 721  19  40   4   8   0   0]
 [  1  10   0  11 738  11  26   3   0   0]
 [  2   2   0   7   5 757  24   3   0   0]
 [  1   1   0   0  11   6 778   0   0   3]
 [  1   7   0   5   4   6   3 774   0   0]
 [  4   1   0   0   0   1   0   0 793   1]
 [ 16   1  12   0   0   0   4   0   5 762]]
 accuracy:0.95675
```