فصل ۱۰

شبکه عصبی پیچشی - Numpy

اهداف این جلسه

شما در این جلسه یاد خواهید گرفت که :

- مراحلِ پیچش [۱] را توضیح دهید.
- دو نوع مختلف از عملیات تجمیع [۲] را پیاده سازی کنید.
- اجزای استفاده شده در یک شبکه عصبیِ پیچشی [۳] (مانند لایه‌گذاری [۴] ، گام [۵] ، فیلتر و ...) و دلیل استفاده از آنها را بشناسید.
- یک شبکه عصبیِ پیچشی بسازید

---

۱ convolution
۲ pooling
۳ Network Neural Convolutional
۴ padding
۵ stride

نشانه گذاری:

- نشانه‌ی $[l]$ نشان‌دهنده‌ی شیء ای است که در لایه ی l ـاُم حضور دارد.
- نشانه‌ی (i) نشان‌دهنده ی شیء ای در نمونه‌ی i ـاُم است.
- نشانه‌ی i نشان‌دهنده ی i ـاُمین ورودیِ بردار است.
- نشانه های n_H, n_W, n_C به ترتیب نشان‌دهنده ی طول، عرض و تعداد کانال های یک لایه ی داده شده می باشند. اگر بخواهیم که به یک لایه ی خاص اشاره کنیم، می‌توانیم از نشانه‌گذاریِ $n_H^{[l]}, n_W^{[l]}, n_C^{[l]}$ استفاده کنیم.
- نشانه های $n_{H_{prev}}, n_{W_{prev}}, n_{C_{prev}}$ نیز به ترتیب طول، عرض و تعداد کانال های لایه ی قبلی را نشان می دهند. اگر بخواهیم به یک لایه ی خاص اشاره کنیم میتوانیم از $n_H^{[l-1]}, n_W^{[l-1]}, n_C^{[l-1]}$ استفاده کنیم.

توجه داشته باشید که شما در این جلسه باید بر کتابخانه ی `numpy` تسلط نسبی داشته باشید.

۱.۱۰ پکیج ها

در قسمت اول، باید پکیج هایی که در این جلسه استفاده خواهید کرد را فراخوانی کنید.

- `NumPy` پکیج اصلیِ محاسباتِ علمی در پایتون است.
- `matplotlib` یک کتابخانه برای مصوّرسازی نتایج در پایتون است.
- از `np.random.seed(1)` برای هماهنگ نگه‌داشتنِ تمام توابع تصادفی استفاده خواهیم کرد. این کار به درجه‌بندی کارِ شما کمک خواهد کرد.

کد مربوط به این قسمت را می‌توانید در فایلِ ژوپیترِ مربوط به این جلسه، در قسمتِ **Packages - 1** مشاهده کنید.

۲.۱۰ دورنمای تمرین های این جلسه

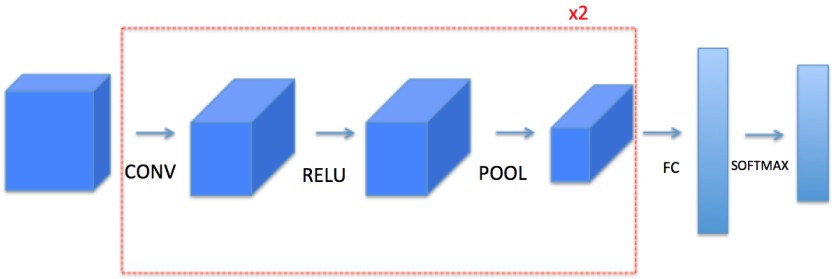
در ادامه شما بلوک هایی از شبکه عصبیِ پیچشی را خواهید ساخت. هر تابعی که پیاده سازی می‌کنید، راهنماهایی برای کمک کردن به شما برای گذراندن مراحلِ زیر خواهد داشت:

- توابعِ پیچشی شاملِ
  - لایه‌گذاری با صفر [۱]
  - پنجره‌ی پیچش [۲]
  - پیچشِ پیش‌رو [۳]
  - پیچش پس‌رو [۴]
- توابعِ تجمیع شاملِ:

[۱] zero padding
[۲] convolve window
[۳] convolution forward
[۴] convolution backward

- تجمیع پیش‌رو [۱]
- ساختِ ماسک
- مقدارِ توزیع [۲]
- توزیع پس‌رو [۳]

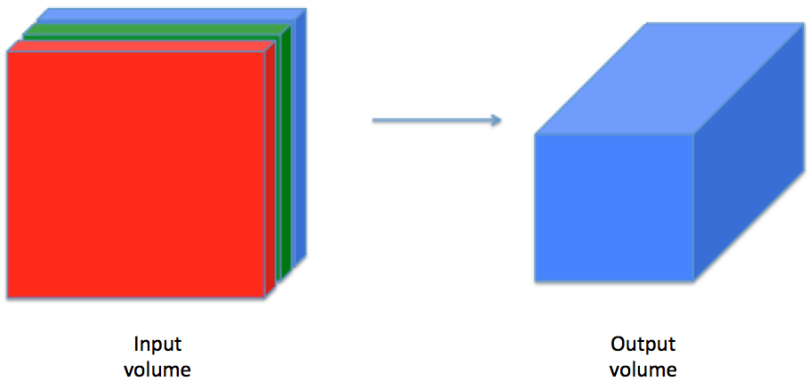
نکته: در این جلسه شما تمامی موارد بالا را با استفاده از `NumPy` ، از صفر تا صد پیاده سازی خواهید کرد. در جلسه ی بعدی، موارد مشابه این توابع را در `TensorFlow` ، برای پیاده سازیِ مدل زیر، یاد خواهید گرفت:



نکته: برای هر تابعِ پیش‌رو [۴] یک تابعِ پس‌رو [۵] ی متناظر نیز وجود دارد . به همین دلیل، در هر قدم از مرحله‌ی پیش‌رو ، شما برخی از پارامتر ها را در یک حافظه ی کوتاه مدت ذخیره‌سازی خواهید کرد. این پارامترها برای محاسبه‌ی گرادیان‌ها در حینِ انتشارِ پس‌رو [۶] استفاده خواهند شد.

۳.۱۰ شبکه‌های عصبیِ پیچشی

اگر چه چهارچوب های برنامه نویسی، [۷] باعث ساده شدنِ استفاده از پیچش ها شده اند، اما آنها هنوز هم یکی از مفاهیم سخت در یادگیری عمیق محسوب می‌شوند. یک لایه ی پیچش ، همانطور که در شکل زیر نشان داده شده است، یک مقدار ورودی را به یک مقدار خروجی با اندازه ای متفاوت تبدیل می‌کند. در این قسمت، شما هر مرحله از لایه ی پیچش

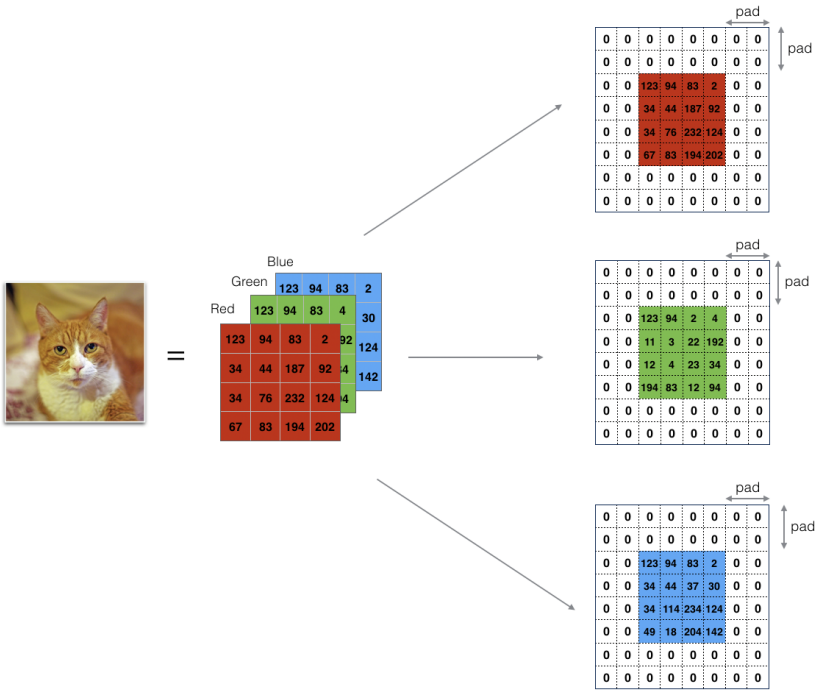


[۱] pooling forward
[۲] Distribute value
[۳] pooling backward
[۴] forward
[۵] backward
[۶] backpropagation
[۷] frameworks programming

را خواهید ساخت. در ابتدا دو تابعِ کمک کننده [۱] : یکی برای لایه‌گذاری با صفر و دیگری برای محاسبه ی تابعِ پیچش را پیاده سازی خواهید کرد.

۱.۳.۱۰ لایه‌گذاری با صفر

لایه‌گذاری با صفر، (شکل ۱) لایه ای از مقادیر صفر را در مرز های یک تصویر، اضافه می‌کند: فواید اصلیِ لایه‌گذاری در



شکل ۱.۱۰: تصویر سه کاناله، همراه با لایه‌ای با مقدارِ ۲

ذیل توضیح داده شده است:

- این کار به ما اجازه می‌دهد که بدون نیاز به کاهش دادنِ طول و عرضِ مقادیر، از یک لایه‌ی CONV استفاده کنیم. این عمل برای ساختنِ شبکه های عمیق تر، اهمیت پیدا می کند زیرا در غیر این صورت، طول و عرض به نسبتِ رفتن به لایه های عمیق‌تر، کاهش پیدا خواهند کرد.
- این کار به ما کمک می‌کند که اطلاعاتی که در مرز های تصاویر وجود دارند را حفظ کنیم. بدونِ لایه‌گذاری ، مقادیر بسیار کمی در لایه‌ی بعدی، از پیکسل های مرزیِ تصویر، تاثیر می پذیرند.

تمرین اول : zero_pad

در فایلِ ژوپیترِ این جلسه، در قسمتِ Exercise 1 - zero_pad، تابعِ `zero_pad(X,pad)` را که تمام تصاویرِ یک دسته مانندِ `X` را، با صفر، لایه‌گذاری می کند، پیاده سازی کنید. در این جا باید از np.pad استفاده کنید. این نکته را در نظر داشته باشید که اگر لایه‌ای با `pad = 1` برای بُعد دوم را در آرایه‌ی "a" با شکلِ (۵،۵،۵،۵) و همچنین `pad = 3` برای بُعد چهارم و `pad = 0` برای بقیه، بگذارید، می‌توانید به صورت زیر عمل کنید:

[۱] helper function

```
a = np.pad(a, ((0,0), (1,1), (0,0), (3,3),(0,0)), mode='constant', constant_values = (0,0))
```

۲.۳.۱۰ یک مرحله از پیچش

در این قسمت، شما باید یک مرحله از پیچش را پیاده‌سازی نمایید که در واقع در آن، فیلتر را بر روی یک قسمت از ورودی اعمال می‌کنید. این کار برای ساخت یک واحدِ پیچشی که :

- یک مقدار ورودی را می‌گیرد.
- یک فیلتر را بر هر قسمت از ورودی اعمال می‌کند.
- یک مقدار دیگر (که معمولا اندازه‌ی آن با اندازه ورودی متفاوت است) را به عنوان خروجی می‌دهد.

استفاده خواهد شد.
در کاربردهای بینایی کامپیوتر، هر مقدار در ماتریس سمت چپ، متناظرِ یک تک مقدارِ پیکسل خواهد بود. شما یک فیلترِ 3x3 را با تصویر، به‌هم‌پیچش [۱] خواهید کرد، به این صورت که مقادیر آن را به صورت درایه ای با ماتریس اصلی، ضرب خواهید کرد و سپس آن‌ها را با هم جمع کرده و در نهایت یک بایاس به آن اضافه خواهید نمود. شما یک مرحله از پیچش را پیاده‌سازی خواهید نمود که متناظر است با اعمال کردنِ یک فیلتر بر روی فقط یک قسمت از تصویر و گرفتنِ یک خروجیِ تک مقداری.
در قسمت های بعدی، شما این تابع را بر روی قسمت های دیگر ورودی نیز اعمال خواهید کرد تا در نهایت یک عملِ پیچش کامل را پیاده‌سازی کرده باشید.

تمرین دوم: conv_single_step

در فایل این جلسه، در قسمتِ Exercise 2-conv_single_step، تابعِ `conv_single_step()` را پیاده‌سازی کنید.
راهنمایی **نکته:** متغیرِ b به صورتِ یک بردارِ numpy منتقل خواهد شد. اگر شما یک مقدارِ اسکالر را با یک بردارِ numpy جمع کنید، خروجی یک بردارِ numpy خواهد بود. در مورد خاصی که بردارِ numpy حاوی یک تک مقدار باشد، شما می‌توانید آن را با cast کردن به یک مقدارِ float ، به مقدارِ اسکالر تبدیل کنید.

۳.۳.۱۰ شبکه‌های عصبی پیچشی - گذرِ پیش‌رو

در گذرِ پیش‌رو [۲] شما تعداد زیادی از فیلترها را گرفته و آنها را بر روی ورودی، اعمال خواهید کرد. هر پیچش، به شما یک ماتریس دو بُعدی به عنوان خروجی می‌دهد. سپس شما باید این خروجی ها را بر روی هم قرار داده و یک مقدارِ سه بُعدی را بسازید :
تصویر پنج

[۱] convolve
[۲] Forward Pass

تمرین سوم: conv_forward

در فایل این جلسه، در قسمتِ Exercise 3-conv_forward، تابعِ
conv_forward(A_prev, W, b, hparameters) را به منظورِ به‌هم‌پیچشِ فیلتر های W بر روی یک ورودیِ فعالسازِ A_prev ، پیاده‌سازی کنید. این تابع ورودی های زیر را دریافت می‌کند:

- A_prev ، فعالسازهای خروجی ای که از لایه ی قبلی حاصل شده اند. (برای دسته ای از m ورودی)
- وزن هایی که با W نشان داده شده اند و اندازه ی فیلتر، f در f است.
- بردارِ بایاسِ b ، که هر فیلتر بایاس مخصوص به خودش را دارد.

شما همچنین به هایپرپارامتر هایی مانند stride و padding نیز دسترسی دارید.

راهنمایی:

- برای انتخابِ یک قطعه‌ی ۲x۲ از گوشه ی بالا سمت چپِ ماتریسِ a_prev ، (ابعاد: (۵،۵،۳)) ، شما می‌بایست مانند قطعه کد زیر عمل کنید:

```
a_slice_prev = a_prev[0:2,0:2,:]
```

توجه کنید که این‌کار، یک تکه‌ی سه بُعدی که ارتفاع آن ۲، عرض آن ۲ و عمق آن ۳ است را به ما می‌دهد که عمق همان تعداد کانال‌ها می‌باشد. هنگامی که a_slice_prev را تعریف می‌کنید، استفاده از نمایه‌های [۱] start/end سودمند خواهد بود.

- برای تعریف کردنِ a_slice احتمالا در ابتدا به تعریفِ گوشه های آن : vert_start ، vert_end ، horiz_start و horiz_end نیاز پیدا خواهید کرد. (شکل ۲) به شما کمک خواهد کرد که گوشه ها را با استفاده از h ، w، f و s در کد مربوط به این قسمت، پیدا کنید.

یادآوری:
فرمول هایی که مربوط به قالب خروجیِ پیچش نسبت به قالب ورودیِ آن هستند، در ذیل آمده است:

$$n_H = \left\lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{\text{stride}} \right\rfloor + 1$$

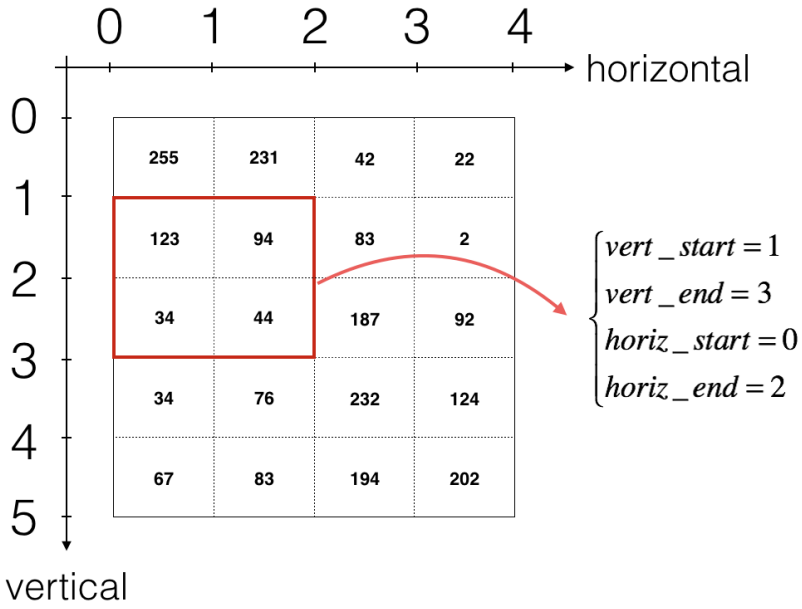
$$n_W = \left\lfloor \frac{n_{W_{\text{prev}}} - f + 2 \times pad}{\text{stride}} \right\rfloor + 1$$

$$n_C = \text{تعداد فیلتر های استفاده شده در کانولوشن}$$

راهنمایی:

- از تکه کردنِ بردارها (مثلا varname[0:1,:,3:5]) برای متغیرهای a_prev_pad ، W و b استفاده کنید.

[۱] indexes



شکل ۲.۱۰: تعریفِ یک تکه با استفاده از آغاز /پایانِ عمودی و افقی با یک فیلترِ ۲x۲ ، این تصویر یک تک کانال را نشان می‌دهد.

- کد شروعِ تابع را کپی و آن را خارج از تعریف تابع در یک سلول جداگانه، اجرا کنید
- بررسی کنید که زیرمجموعه‌ی هر آرایه و بُعد آن، همانی باشد که انتظارش را دارید.

• برای اینکه تصمیم بگیرید که

`vert_start` ، `vert_end` ، `horiz_start` و `horiz_end` را چگونه بدست بیاورید، یادتان باشد که این‌ها، نمایه‌های لایه‌ی قبلی هستند.

- یک نمونه از لایه‌ی لایه‌گذاری شده‌ی قبلی (برای مثال ۸x۸) و لایه‌ی کنونی (لایه‌ی خروجی) (برای مثال ۲x۲) را ترسیم کنید.
- نمایه‌های لایه‌ی خروجی با `h` و `w` نشان داده شده اند.

• مطمئن شوید که `a_slice_prev` ارتفاع، عرض و عمق دارد

• یادتان باشد که `a_prev_pad` زیرمجموعه‌ای از `A_prev_pad` است.

- به این فکر کنید که از کدامیک باید در حلقه‌ها استفاده کنید.

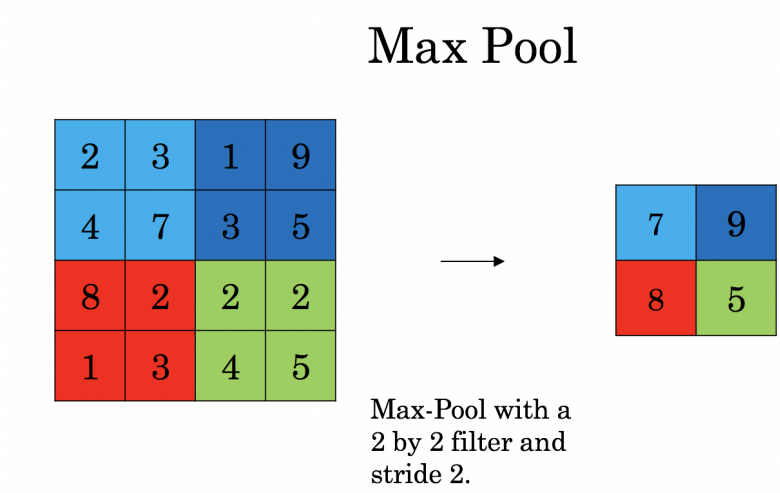
در نهایت یک لایه‌ی CONV همچنین باید شاملِ یک فعال ساز باشد، پس شما می‌توانید قطعه‌ی زیر را به کد خود اضافه کنید:

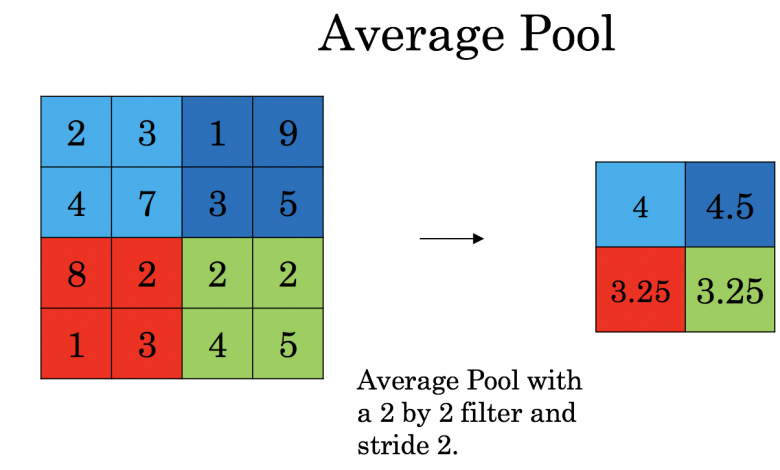
```
# Convolve the window to get back one output neuron
Z[i, h, w, c] = ...
# Apply activation
A[i, h, w, c] = activation(Z[i, h, w, c])
```

۴.۱۰ لایه‌ی تجمیع

وظیفه‌ی لایه‌ی تجمیع (POOL) ، کاهشِ طول و عرضِ ورودی است. با این کار، حجم محاسبات کاهش می‌یابد و همچنین کمک می کند تا آشکارسازهای ویژگی با موقعیت خود در ورودی تغییرناپذیرتر شوند
دو نوع لایه‌ی تجمیع عبارتند از:

- لایه Max-pooling: یک پنجره‌ی (f, f) را بر روی ورودی حرکت می‌دهد و مقدار بیشینه‌ی داخل پنجره را در خروجی ذخیره می‌کند.
- لایه‌ی Average-pooling: یک پنجره ی (f, f) را بر روی ورودی حرکت می‌دهد و مقدار متوسط مقادیر زیر پنجره را در خروجی ذخیره می‌کند





این لایه ها، هیچ پارامتری برای استفاده در انتشارِ پس‌رو برای آموزش ، ندارند.

۱.۴.۱۰ تجمیعِ پیش‌رو

در این قسمت می‌خواهیم در یک تابع، MAX-POOL و AVG-POOL را پیاده سازی کنیم.

تمرین چهارم: pool_forward

در فایل ژوپیتر مربوط به این جلسه، در قسمت Exercise 4-pool_forward، گذرِ پس‌رو را برای لایه‌ی تجمیع، پیاده سازی کنید. از راهنمایی هایی که در قسمت کامنت های کد نوشته شده است، استفاده کنید.
یادآوری: از آنجایی که padding نداریم، فرمول هایی که شکل ورودی را به شکل خروجی متصل می‌کنند، به صورت زیر تعریف می‌شوند:

$$n_H = \left\lfloor \frac{n_{H_{\text{prev}}} - f}{\text{stride}} \right\rfloor + 1$$

$$n_W = \left\lfloor \frac{n_{W_{prev}} - f}{\text{stride}} \right\rfloor + 1$$

$$n_C = n_{C_{prev}}$$

خروجیِ مورد انتظار در حالت اول:

```
Case 1: stride of 1
mode = max
A.shape = (2, 3, 3, 3)
A[1, 1] =
 [[1.96710175 0.84616065 1.27375593]
 [1.96710175 0.84616065 1.23616403]
 [1.62765075 1.12141771 1.2245077 ]]

mode = average
A.shape = (2, 3, 3, 3)
A[1, 1] =
 [[ 0.44497696 -0.00261695 -0.31040307]
 [ 0.50811474 -0.23493734 -0.23961183]
 [ 0.11872677  0.17255229 -0.22112197]]
```

خروجی مورد انتظار در حالت دوم:

```
Case 2: stride of 2
mode = max
A.shape = (2, 2, 2, 3)
A[0] =
 [[[1.74481176 0.90159072 1.65980218]
  [1.74481176 1.6924546  1.65980218]]

 [[1.13162939 1.51981682 2.18557541]
  [1.13162939 1.6924546  2.18557541]]]

mode = average
A.shape = (2, 2, 2, 3)
A[1] =
 [[[-0.17313416  0.32377198 -0.34317572]
  [ 0.02030094  0.14141479 -0.01231585]]

 [[ 0.42944926  0.08446996 -0.27290905]
  [ 0.15077452  0.28911175  0.00123239]]]
```

۵.۱۰ انتشارِ پس‌رو در شبکه‌های عصبیِ پیچشی

در چهارچوب‌های مدرنِ یادگیری عمیق ، شما فقط نیاز دارید که گذرِ پیش‌رو را پیاده سازی کنید و خودِ چهارچوب، به قسمت گذرِ پس‌رو [۱] رسیدگی می‌کند. پس اکثر مهندسین یادگیری عمیق، نیازی به عمیق شدن در مرحله‌ی گذرِ پس‌رو ندارند. گذرِ پس‌رو برای شبکه های پیچشی، مرحله ای پیچیده است. در قسمت بعدی قصد داریم با مرحله‌ی انتشار پس‌رو در یک شبکه‌ی پیچشی، آشنا شویم.
در شبکه‌های پیچشی، شما می‌توانید مشتقات را با توجه به هزینه، برای به‌روزرسانیِ پارامترها، استفاده کنید.

۱.۵.۱۰ لایه‌ی پیچشی - گذرِ پس‌رو

در این قسمت می‌خواهیم برای لایه‌ی CONV ، گذرِ رو به عقب را پیاده سازی نماییم.

محاسبه‌ی dA

فرمول محاسبه‌ی dA با داشتنِ هزینه برای یک فیلتر خاصِ $\boldsymbol{W}_c$ و نمونه آموزشیِ داده شده، بصورت زیر می‌باشد:

$$dA+ = \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw} \tag{۱.۱۰}$$

که در آن W_c فیلتر و dZ_{hw} مقدار اسکالرِ متناظر با گرادیانِ هزینه با توجه به خروجیِ لایه‌ی کانولوشنِ Z در سطرِ h ـاُم و ستونِ w ـاُم می‌باشد. نکته آن که در هنگام به‌روزرسانیِ dA در هر دفعه، شما همان فیلترِ W_c را در مقدار متفاوتی از dZ ضرب می‌کنید. این کار بیشتر برای آن انجام می‌شود که در هنگامِ محاسبه‌ی انتشارِ پیش‌رو، هر فیلتر توسط یک

[۱] backward pass

a_slice متفاوت ضرب داخلی و جمع می‌شود. به همین جهت، هنگام محاسبه‌ی انتشار پس‌رو برای dA شما فقط گرادیان‌ها را به همه‌ی a_slice ها اضافه می‌کنید. درکد، داخل حلقه‌های for مناسب، این فرمول، به صورت زیر نوشته می‌شود:

```
da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] += W[:,:,:,c] * dZ[i, h, w, c]
```

محاسبه‌ی dW

فرمولِ محاسبه‌ی dW_C (مقدار متناظر مشتقِ یک فیلتر) با توجه به هزینه، بصورت:

$$dW_c+ = \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw} \tag{۲.۱۰}$$

می‌باشد که a_{slice} به slice ای مربوط است که هنگامِ ایجادِ فعال‌سازِ Z_{ij} استفاده شده است. از این رو، در نهایت ما گرادیانِ W را با توجه به آن slice خواهیم داشت. از آنجایی که این همان W است، ما فقط همه‌ی گرادیان ها را با هم جمع می‌کنیم تا dW را بدست بیاوریم.
درکد، در حلقه‌های for مناسب، این فرمول بصورت زیر نوشته می‌شود:

```
dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
```

محاسبه‌ی db

در ذیل فرمول محاسبه‌ی db را با توجه به هزینه برای یک فیلترِ خاصِ W_c داریم:

$$db = \sum_{h} \sum_{w} dZ_{hw} \tag{۳.۱۰}$$

همانطورکه می‌دانید، db با جمع کردنِ dZ بدست می‌آید. در این مورد، شما صرفا در حال جمع کردن بر روی تمام گرادیان های خروجیِ کانولوشن (Z) با توجه به هزینه، هستید.
درکد، در حلقه‌های for مناسب، این فرمول به صورت زیر نوشته می‌شود:

```
db[:,:,:,c] += dZ[i, h, w, c]
```

تمرینِ پنجم conv_backward

در فایل این جلسه، در قسمتِ Exercise 5-conv_backward، تابعِ `conv_backward` را تکمیل کنید. شما باید بر روی تمامی نمونه‌های آموزشی، فیلترها، طول و عرض‌ها، عمل جمع را انجام دهید. سپس باید مشتقات را با استفاده از فرمول‌هایِ ۱ ، ۲ و ۳ که در بالا نوشته ایم، بدست آورید

خروجی مورد انتظار:

```
dA_mean =1.45243777754
dW_mean =1.72699145831
db_mean =7.83923256462
```

۲.۵.۱۰ لایه‌ی تجمیع - گذرِ پس‌رو

در ادامه ، گذرِ پس‌رو را برای لایه‌ی تجمیع پیاده سازی خواهیم کرد. برای این کار از لایه‌ی MAX-POOL شروع خواهیم کرد. اگرچه یک لایه‌ی تجمیع ، هیچ پارامتری برای به‌روزرسانی هنگام انتشارِ پس‌رو ندارد، شما همچنان نیاز دارید که گرادیان را در طول لایه‌ی تجمیع ، انتشارِ پس‌رو کنید تا بتوانید گرادیان‌ها را برای لایه‌هایی که قبل از لایه‌ی تجمیع می‌آیند، محاسبه کنید.

تجمیعِ بیشینه - گذرِ پس‌رو

قبل از آنکه سراغ انتشارِ پس‌رو برای لایه‌ی تجمیع برویم، شما باید تابعِ کمکیِ `create_mask_from_window()` را بسازید که بصورت زیر کار می‌کند:

$$X = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \quad \rightarrow \quad M = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{۴.۱۰}$$

همانطور که میبینید، این تابع یک ماتریسِ mask تولید می‌کند تا مشخص کند که مقدار بیشینه‌ی ماتریس را در کجا نگهداری کند. مقدارِ صحیح (۱) محل مقدار بیشینه را در ماتریسِ X معین می‌سازد و بقیه مقادیر، صفر هستند.

پیاده سازیِ create_mask_from_window

در فایل مربوط به این جلسه، در قسمتِ Exercise 6 - create_mask_from_window، تابعِ `create_mask_from_window()` را پیاده‌سازی کنید. این تابع در هنگام تجمیعِ پس‌رو، مفید واقع خواهد شد.

- np.max() ممکن است مثمر ثمر واقع شود، این تابع، بیشینه‌ی یک بردار را محاسبه می‌کند.
- اگر شما ماتریسِ X و مقدار اسکالرِ x را داشته باشید، `A = (X == x)` یک ماتریسِ A با اندازه ای مشابهِ X را بر می‌گرداند، به طوری که:

$$A[i,j] = \text{ if True } X[i,j] = x$$
$$A[i,j] = \text{ if False } X[i,j] != x$$

- در اینجا شما نیاز نیست که مواردی که در یک ماتریس، تعداد بیشینه ها بیشتر از یک عدد باشد را در نظر بگیرید.

خروجی مورد انتظار:

```
x =
[[ 1.62434536 -0.61175641 -0.52817175]
[-1.07296862 0.86540763 -2.3015387 ]]
mask = [[ True False False]
[False False False]]
```

چرا موقعیت بیشینه را ردیابی کنیم؟ زیرا این مقدار ورودی است که در نهایت بر خروجی و در نتیجه هزینه تأثیر می گذارد. انتشارِ پس‌رو، محاسبه‌ی گرادیان‌ها با توجه به هزینه، است، بنابراین هر چیزی که بر هزینه نهایی تأثیر می گذارد باید دارای گرادیانِ غیر صفر باشد. پس انتشارِ پس‌رو، گرادیان را به سمت این مقدار خاص که بر روی هزینه تاثیر گذاشته است، رو به عقب، منتشر می‌کند.

تجمیعِ متوسط - گذرِ پس‌رو

در تجمیعِ بیشینه، برای هر پنجره‌ی ورودی، تمامی تاثیراتِ روی خروجی، از یک مقدارِ ورودی که همان مقدار بیشینه بود، می‌آمد. در تجمیعِ متوسط [1] ، هر جزء از پنجره‌ی ورودی، تاثیر مشابهی نسبت به بقیه بر روی خروجی دارد، پس برای پیاده سازیِ پس‌رو ، شما ابتدا باید تابع کمکی‌ای که این امر را بازتاب دهد، پیاده سازی کنید.
برای مثال، اگر ما بر روی گذرِ پیش‌رو، تجمیعِ متوسط را با استفاده از یک فیلترِ ۲x۲ انجام بدهیم، آنگاه ماسکی که برای گذرِ پس‌رو استفاده خواهید کرد، به صورت زیر، درخواهد آمد:

$$dZ = 1 \quad \rightarrow \quad dZ = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix} \tag{۵.۱۰}$$

نتیجه آنکه هر نقطه در ماتریسِ ،dZ به طور مساوی بر روی خروجی، تاثیر خواهد گذاشت.

۳.۵.۱۰ تمرین هفت distribute_value

در فایل این جلسه، در قسمتِ Exercise 7-distribute_value، تابعِ `distribute_value(dz, shape)` را تکمیل کنید. این تابع به طور مساوی، مقدارِ dZ را بر روی مقادیر ماتریسی با بَعدِ `shape` پخش خواهد کرد. راهنمایی

خروجی مورد نظر:

```
distributed_value = [[ 0.5 0.5]
[ 0.5 0.5]]
```

جمع بندیِ نهایی : تجمیعِ پس‌رو

اکنون شما تمامی پیشنیازها را برای محاسبه‌ی پخشِ پس‌رو بر روی یک لایه‌ی تجمیع در اختیار دارید.

تمرین هشتم pool_backward

در فایل این جلسه، در قسمتِ Exercise 8-pool_backward تابعِ `pool_backward` را در هر دو حالتِ `"max"` و `"average"` پیاده سازی کنید. شما از ۴ حلقه‌ی for برای گذر در طول نمونه های آموزشی، طول، عرض و کانال ها، استفاده خواهید کرد. شما باید از یک متغیر شرطیِ `if/elif` برای کنترل بین مقدار بیشینه یا مقدار متوسط، استفاده کنید. اگر این مقدار، برابر مقدار متوسط بود، باید از تابعِ `distribute_value()` که در تمرین های قبلی پیاده سازی کردید، برای ساختن یک ماتریس با شکلی مشابه با `a_slice` استفاده کنید. در غیر این صورت،

[1] Average Pooling

حالت بر روی `max` است و شما باید یک ماسک با استفاده از `create_mask_from_window()` استفاده کنید و آن را در مقدار متناظرِ dA ضرب کنید.

خروجی مورد نظر:

```
mode = max:
mean of dA =
0.145713902729
dA_prev[1,1] = [[ 0. 0. ]
[ 5.05844394 -1.68282702]
[ 0. 0. ]]

mode = average
mean of dA =
0.145713902729
dA_prev[1,1] = [[ 0.08485462 0.2787552 ]
[ 1.26461098 -0.25749373]
[ 1.17975636 -0.53624893]]
```