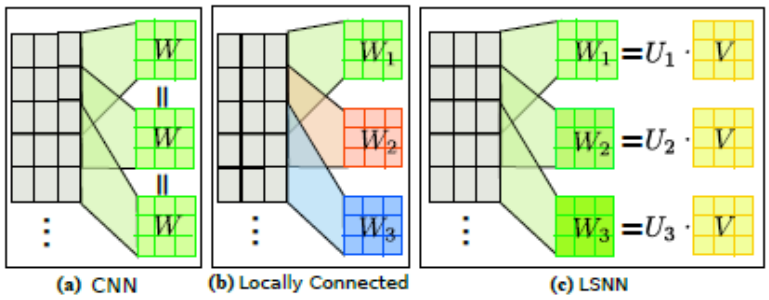
-1

شبکه‌های عصبی Convolutional (CNN) و Locally connected layer در ثبت و در نظر گرفتن اهمیت و روابط بین local receptive fieldهای (دامنه‌های تاثیر محلی) مختلف محدود هستند که این برای کارهایی مانند face verification (تایید چهره) ، visual question answering (پرسش و پاسخ از تصویر) و word sequence prediction (پیشبینی ترتیب کلمه) اساسی و ضروری می‌باشد.

مکانیزم weight sharing در CNN محدودیت خود را در تفاوت قائل شدن برای اهمیت local receptive fieldهای مختلف نشان داده است. Locally connected layer که وزن‌ها را به صورت مستقل به هر local receptive field نسبت می‌دهد و از weight sharing استفاده نمی‌کند نیز به توانایی تعمیم (generalization) آسیب می‌رساند چون در مدل کردن ارتباط بین چندین local receptive field شکست می‌خورد.

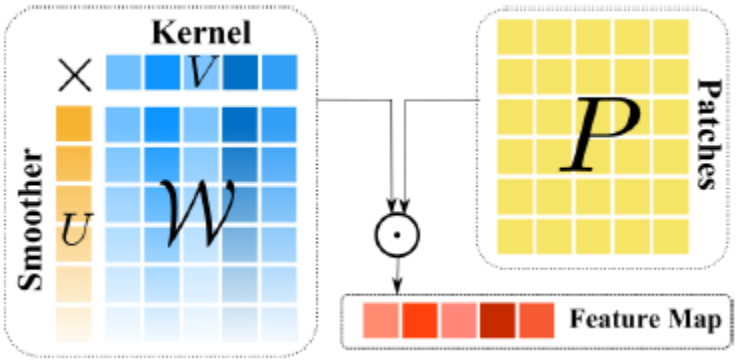
برای حل این مشکل یک شبکه عصبی جدید به نام locally smoothed neural network (LSNN) را معرفی می‌شود. ایده‌ی اصلی این است که ماتریس وزن در Locally connected layer را به صورت ضرب kernel در smoother بیان کنیم که kernel یک بردار هست که بین local receptive fieldهای مختلف به اشتراک گذاشته می‌شود و مفهومی مشابه kernel در CNN دارد. smoother یک بردار است که اهمیت local receptive fieldهای مختلف را در مکان متناظر نشان می‌دهد. برای ایجاد smoother یک تابع گاوسی چند متغیره به کار گرفته می‌شود تا روابط مکانی میان local receptive fieldهای مختلف را مدل کنیم. علاوه بر این از اطلاعات محتوایی می‌توان با تنظیم متغیر mean (میانگین) و precision (صحت) بر طبق محتوی بهره برد. برای مثال یک شبکه عصبی با لایه‌ی رگرسیون fully connected را می‌توان به کار برد تا پارامترهای گاوسی را از روی داده‌ی ورودی ایجاد کرد.

مشاهدات نشان می‌دهد که LSNN رابطه‌ی نزدیکی با CNN و Locally connected layer دارد. CNN را می‌توان یک LSNN با یک smoother شامل برداری از یک‌ها در نظر گرفت. از سوی دیگر Locally connected layer را می‌توان یک LSNN با smoother مستقل در نظر گرفت. علاوه بر این LSNN هم‌ارز با Locally connected layer در نظر گرفت که مرتبه ماتریس وزن در آن برابر یک هست. بنابراین LSNN یک تعادلی بین Locally connected layer و CNN ایجاد می‌کند. در شکل زیر ارتباط بین سه شبکه را می‌بینید.



محدودیت weight sharing در CNN بسیار سختگیرانه است و فرض مستقل بودن پارامترها در local receptive fieldهای مختلف در Locally connected layer بسیار سهل‌انگارانه هست. بنابراین یک ایده این است که از kernelهای وزن‌دار بر روی local receptive fieldهای مختلف استفاده کرد.

ساختار LSNN به صورت زیر هست:



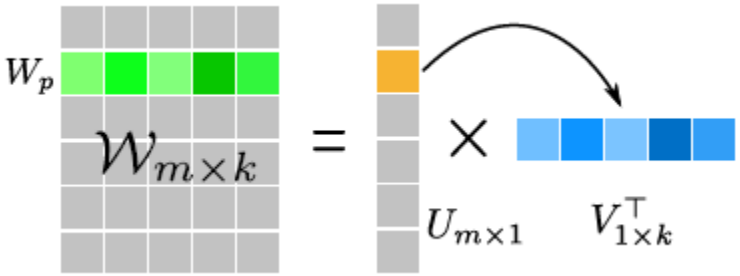
ابتدا همه‌ی پارامترهای وزن در local receptive fieldهای مختلف را در کنار یکدیگر قرار می‌دهیم تا ماتریس وزن را بسازیم: $\mathcal{W} = [W_p]_{p \in \mathcal{P}}$, که W_p نشان دهنده بردار kernel در مکان‌های مختلف p هست و P نشان‌دهنده تمام مکان‌های ممکن است. به عنوان مثال اگر بردار ورودی یک بعدی باشد اگر بردار ورودی به شکل $X = [x_1, x_2, \ldots, x_n]$, باشد P_i با k بعد به صورت :

$$P_i = [x_i, x_{i+1}, \ldots, x_{i+k-1}]^\top, i = 1, \cdots, m \text{ where } m = n - k + 1.$$

تعریف می‌شود و kernel متناظر را با W_i نشان می‌دهیم. بنابراین ماتریس وزن را می‌توانیم به صورت زیر می‌باشد:

$$\mathcal{W} = [W_1, W_2, \cdots, W_m]^\top.$$

تجزیه ماتریس وزن در LSNN به صورت زیر می‌باشد:



$$\mathcal{W} = UV^\top, \text{ i.e. } W_p = U_p V^\top, \forall p \in \mathcal{P},$$

بردار U همان smoother می‌باشد که در آن هر عضو نشان‌دهنده اهمیت فیلترهای متفاوت هست. بردار V نیز همان بردار kernel هست.

برای گرفتن "کجا" و "چه چیزی" از اطلاعات داده smoother گاوسین را معرفی می‌کنیم. تابع گاوسی با دو پارامتر تعریف می‌شود: mean (میانگین) μکه نشان‌دهنده موقعیت هست و precision (صحت) **Λ** که ناحیه را کنترل می‌کند. پارامترهای گاوسی μ و **Λ** از روی patchهای داده‌ی ورودی تعیین می‌شوند و سپس smoother U را با μ و **Λ** تعیین می‌کنیم.

$$\boldsymbol{\mu} = g(X), \quad \boldsymbol{\Lambda} = h(X), \quad U = f(\boldsymbol{\mu}, \boldsymbol{\Lambda}),$$

که h و g شبکه‌های عصبی feed forward برای تعیین μ و **Λ** هستند و Parameter Network نامیده می‌شوند. تابع f نیز یک تابع گاوسی هست. برای هر مکان p تابع گاوسی U_p به صورت زیر محاسبه می‌شود.

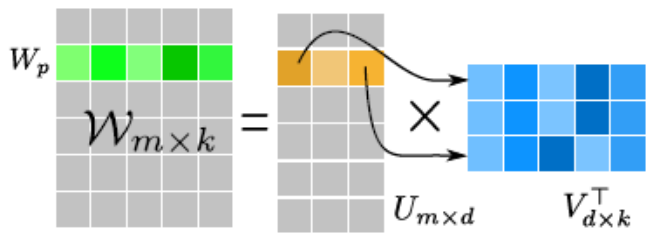
$$U_p = \exp(-(p - \boldsymbol{\mu})^\top \boldsymbol{\Lambda}(p - \boldsymbol{\mu})).$$

اگر μ و **Λ** پارامترهای مستقلی باشند که از داده یاد گرفته می‌شوند smoother می‌تواند ارتباط مکانی بین فیلترهای متفاوت را ثبت کند در غیر این صورت می‌توان اطلاعات محتوایی را به کار برد تا اهمیت فیلترهای متفاوت را تعیین کرد.

برای فرایند بهینه‌سازی از روش back-propagation استاندارد استفاده شده است. برای پیاده‌سازی بهینه‌ساز SGD اتخاب شده است و learning rate بخش Parameter Network را یک دهم دیگر بخش‌های LSNN قرار داده شده است. زیرا پارامترهای smoother تنها بخش کوچکی از کل پارامترهای LSNN هستند.

در اینجا از smoother به دلیل تحلیل و پیاده‌سازی ساده‌تر و ... انتخاب شده است، اما در عمل می‌توان از smoootherهای دیگر نیز استفاده کرد. هر چقدر مسئله پیچیده‌تر باشد می‌توان از تابع پیچیده‌تری به عنوان smoother استفاده کرد.

تا الآن ماتریس وزن را به صورت حاصل ضرب دو بردار می‌نوشتیم که به آن factorization یک بعدی گفته می‌شود، اما می‌توان factorization با ابعاد بالاتر نیز حساب کرد. این نوع از factorization را در شکل زیر می‌بینید.



مکانیزم‌های توجه (attention) به سیستم اجازه می‌دهند تا به صورت ترتیبی بر زیرمجموعه‌های مختلفی از داده‌های ورودی تمرکز کند. اتخاب این زیرمجموعه معمولا به حالت (state) سیستم بستگی دارد که خود تابعی از زیر مجموعه‌ی قبلی است که به آن "توجه" شده‌است. LSNN را می‌توان به عنوام یک مکانیزم soft-attention دید که از یک وزن‌دهی نرم (soft) برای زیر مجموعه‌های مختلف استفاده می‌کند. smoother گاوسی هم اطلاعات مکانی و هم اطلاعات محتوایی را مد نظر قرار می‌دهد که به سیستم اجازه می‌دهد به جنبه‌های مختلف ورودی توجه داشته باشد.

دو نسخه از LSNN وجود دارد. در یکی پارامترهای smoother گاوسی، آزاد هستند که به آن LSNN-Location گفته می‌شود و در دیگری این‌ها پارامترهای شبکه هستند که به آن LSNN-Content می‌گویند.

برای مشاهده‌ی کارایی LSNN بروی چندین نسخه‌ی تغییریافته از MNIST چند آزمایش انجام دادیم:

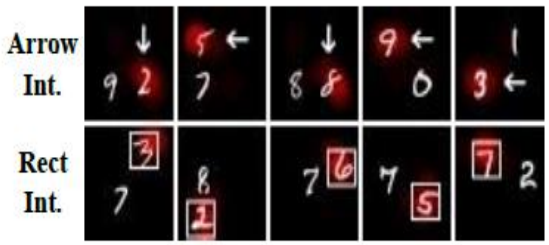
نسخه‌ی اول : در این نسخه یک تصویر 28×28 پیکسل از اعداد MNIST را در یک پس زمینه سیاه 84×84 قرار دادیم و تصاویر crop شده‌ی 6×6 از MNIST اصلی را به عنوان distractor در عکس قرار می‌دهیم و آن عکس را 42×42 resize می‌کنیم.

The error rate on the clutter translated MNIST task.

Model	Error
Convolutional Layer	3.88%
Locally Connected Layer	9.54%
LSNN-Location	3.08%
LSNN-Content	**2.89%**

Locally Connected Layer بسیار حساس به نویز بوده و نتیجه خوبی ندارد چون پارامتر Local receptive fieldهای آن مستقل بوده. CNN نتیجه قابل قبولی دارد و محدویت weight sharing اینجا مانند یک regularization عمل می‌کند. در بین دو شیوه‌ی LSNN همانطور که می‌بینیم LSNN-Content نتیجه بهتری دارد.

نسخه دوم: در این نسخه دو تصویر 28×28 پیکسل از اعداد MNIST را در یک پس زمینه سیاه 84×84 قرار دادیم و عددی که مورد نظرمان هست را با یک علامت (کمان یا مربع به دور عدد) مشخص کردیم و بعد تصویر



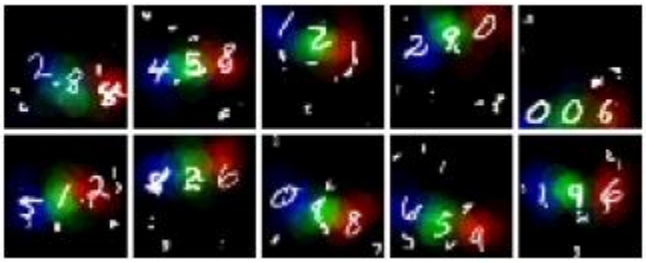
را به 42×42 resize کردیم.

Table 2: The error rate on the intention signed MNIST task.

Model	Arrow Int.	Rect Int.
Convolutional Layer	2.05%	6.25%
Locally Connected Layer	2.53%	13.76%
LSNN-Location	1.60%	4.02%
LSNN-Content	**1.25%**	**3.33%**

همانطور که می‌بینید عملکرد دو شیوه‌ی LSNN از CNN و Locally Connected Layer بسیار بهتر می‌باشد. همچنین LSNN-Content از LSNN-Location عملکرد بهتری دارد به این دلیل که برای تشخیص local receptive fieldهای مهمی که عدد صحیح در آن واقع شده است به smoother بر مبانای محتوی احتیاج داریم.

نسخه سوم: در این نسخه در یک پس زمینه‌ی سیاه 100×100 سه عدد از MNIST قرار گرفته‌اند که مکان اولی به صورت تصادفی انتخاب می‌شود و دوتای دیگر در یک بازه‌ی زاویه‌ای [-45°,45°] نسبت به آن قرار می‌گیرند و 8 عدد crop شده 6×6 نیز در تصویر قرار می‌گیرند. در نهایت تصویر را به 42×42 resize کردیم.



The error rate on the cluttered MNIST sequence task.

Model	Error
Convolutional Layer	23.97%
Locally Connected Layer	13.5%
LSNN-Location	9.98%
LSNN-Content	**6.18%**

CNN بدترین عملکرد را دارد و نتیجه خوبی ندارد، روش‌های LSNN در مقابل بهترین عمکرد را دارند.

آزمایشات فوق نشان دادند که:

1- LSNN توانایی این را دارد که بر concentration (ناحیه‌ی) مورد نظر متمرکز شود.

2- با یک smoother مبتنی بر محتوی وقتی که این concentration مورد نظر به صورت ضمنی با یک علامت مشخص شده باشد LSNN می‌تواند با شناخت علامت و فهمیدن هدف آن علامت به اثر دست یابد.
3- وقتی که concentration توزیع مکانی پیچیده‌ای دارد یعنی چند تا concentration مرتبط به هم وجود دارد LSNN می‌تواند به صورت موثری این توزیع را با استفاده از چند smoother ثبت کند.

-2

الف)

در لایه‌های fully connected هر نورون به تمامی نورون‌های لایه‌ی قبلی متصل است و مثلا اگر روی یک تصویر از یک لایه‌ی fully connected استفاده کنیم یک سری ویژگی و بازنمایی کلی و عمومی از روی تصویر استخراج می‌کند. fully connected برای زمانی مهم است که ترتیب و قرار گرفتن سیگنال‌های داده کنار هم معنا ندارد و می‌توان ویژگی‌ها (features) را جابجا کرد. مثلا وقتی می‌خواهیم از روی یک سری ویژگی مثل سال ساخت، مساحت و ... قیمت خانه‌ها را پیشبینی کنیم ترتیب قرار گرفتن این ویژگی‌ها فاقد معناست و می‌توان آن‌ها را جابجا کرد بنابراین استفاده از fully connected مناسب است.

لایه‌های Convolutional و Locally connected برای استخراج بازنمایی مناسب از تمامی نورون‌های لایه‌ی قبل استفاده نمی‌کنند. این نوع از لایه‌ها بسیار مناسب هستند برای سیگنال‌هایی که به صورت grid هستند. مثلا تصویرها grid هستند یعنی پیکسل‌های مجاور با هم ارتباط دارند و کنار هم بودن و ترتیبشان معنی دارد و چند ویژگی مستقل از هم نیستند. یا مثلا سیگنال‌های صوتی که یک بعدی هستند اما ترتیب و کنار هم بودنش دارای معنا می‌باشد. و همچنین جملات که از کلماتی تشکیل شده که ترتیبشان معنا دارد و اگر جای کلمات عوض شود معنای جمله به هم می‌ریزد. این ارتباط و معنای مجاور بودن سیگنال‌ها در fully connected اهمیتی ندارد.

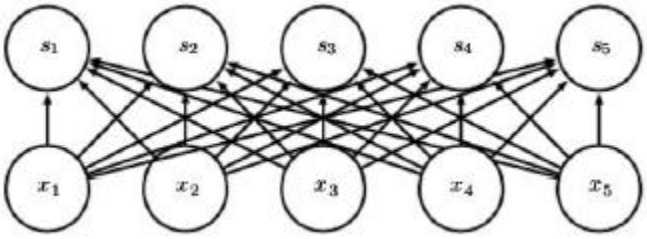
پس در fully connected هر نورون به تمامی نورون‌های لایه‌ی قبل وصل بود، اما در Locally Connected فقط به بخش کوچکی وصل است. خیلی از ویژگی‌هایی که چشم انسان هم detect می‌کند local هستند و به کل تصویر بستگی ندارند. پس در Locally Connected هر نورون می‌تواند یک سری ویژگی ساده را یاد بگیرد.

لایه‌های Convolutional توسعه یافته‌ی لایه‌های Locally connected هستند. فرق Convolutional با Locally connected در این است که فیلترها وزنشان را با یکدیگر به اشتراک می‌گذارند و یا به عبارت دیگر در آن مکانیزم weight sharing وجود دارد. زمانی weight sharing به کار می‌آید که مثلا می‌خواهیم یک مشخصه‌ی ساده را از کل تصویر استخراج کنیم و برایمان مهم نیست بالا یا پایین تصویر باشد. Locally connected مثلا برای

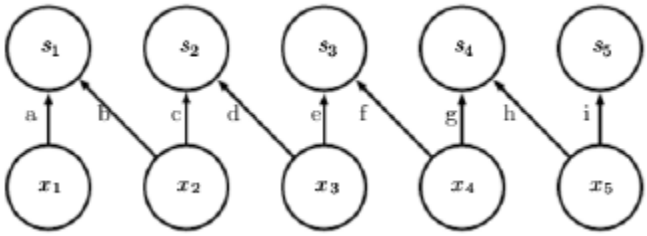
تشخیص لب یا ابرو در تصویر مناسب است چون لب و ابرو در بخش خاصی از اصویر هست و لزومی ندارد وزن‌ها برای آن در همه جا یکسان باشد. به طور کلی در لایه‌های ابتدایی هیچوقت از Locally connected استفاده نمی‌کنیم چون در لایه‌های ابتدایی می‌خواهیم یک سری ویژگی عمومی مثل لبه را در تصویر پیدا کنیم اما در لایه‌های انتهایی که مثلا می‌خواهیم تعیین کنیم که هر تصویرمتعلق به چه کسی است Locally connected layerها خوب است که استفاده کنیم. زمانی که می‌خواهیم تابعی که معادله‌ای که به دست می‌آوریم تابع زمان یا مکان باشد استفاده از Locally connected خوب است.

در fully connected باید سایز ورودی ثابت باشد اما در Convolution و Locally connected سایز ورودی می‌تواند متغیر باشد.

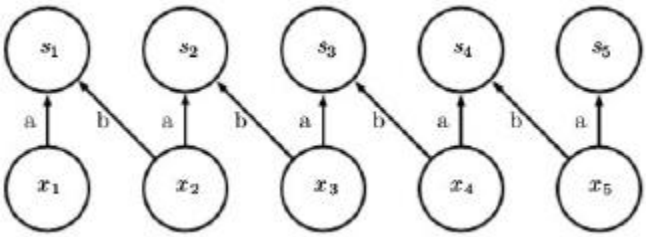
تعداد پارمترهای Convolution از Locally connected کمتر و تعداد پارامترهای Locally connected از fully connected خیلی کمتر است.



Fully Connected



Locally Connected



Convolution

ب)

مزیت‌ها:

1- تعداد پارامترهای شبکه را کاهش می‌دهد و از overfit شدن جلوگیری می‌کند. پارامترهای لایه‌های CNN تغییر نمی‌کند اما خروجی لایه‌ی آخر که flat می‌شود و به fully connected وارد می‌شود. (تعمیم‌دهی بهتر)
2- هزینه‌ی محاسباتی را کاهش می‌دهد. (ویژگی‌های کمتر ولی مفیدتری داریم)
3- ویژگی‌هایی که استخراج می‌کنیم از بخش بزرگتری ازسیگنال باشد بنابراینReceptive field افزایش میابد که منجر به افزایش دقت می‌شود.

معایب:

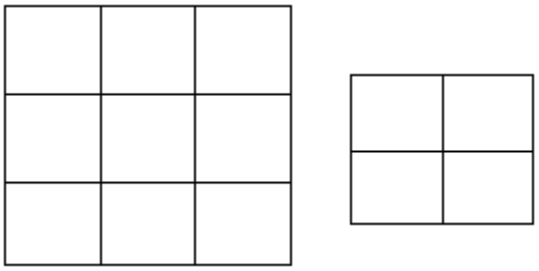
1- حساب نکردن و حذف تعدادی از نورون‌ها باعث می‌شود یک سری از جزئیات را از دست بدهیم که شاید نورون‌های قبلی و بعدی آن نورون‌ها نتوانند آن جزئیات را یاد بگیرند.

مراجع:

اسلایدهای درس

3-

الف)



Forward(input, weight, bias):

output[0][0] = weight[0][0] * input[0][0] + weight[0][1] * input[0][1] +
weight[1][0] * input[1][0] + weight[1][1] * input[1][1] + bias;

output[0][1] = weight[0][0] * input[0][1] + weight[0][1] * input[0][2] +
weight[1][0] * input[1][1] + weight[1][1] * input[1][2] + bias;

```
output[1][0] = weight[0][0] * input[1][0] + weight[0][1] * input[1][1] +
        weight[1][0] * input[2][0] + weight[1][1] * input[2][1] + bias;

output[1][1] = weight[0][0] * input[1][1] + weight[0][1] * input[1][2] +
        weight[1][0] * input[2][1] + weight[1][1] * input[2][2] + bias;
```

ب)

```
Compute_ derivative:
    Result = (weight[0][0] * input[0][0] + weight[0][1] * input[0][1] +
            weight[1][0] * input[1][0] + weight[1][1] * input[1][1] +
            weight[0][0] * input[0][1] + weight[0][1] * input[0][2] +
            weight[1][0] * input[1][1] + weight[1][1] * input[1][2] +
            weight[0][0] * input[1][0] + weight[0][1] * input[1][1] +
            weight[1][0] * input[2][0] + weight[1][1] * input[2][1] +
            weight[0][0] * input[1][1] + weight[0][1] * input[1][2] +
            weight[1][0] * input[2][1] + weight[1][1] * input[2][2] + 4*bias) / 4;

    weight_prime[0][0] = (input[0][0] + input[0][1] + input[1][0] + input[1][1]) / 4;
    weight_prime[0][1] = (input[0][1] + input[0][2] + input[1][1] + input[1][2]) / 4;
    weight_prime[1][0] = (input[1][0] + input[1][1] + input[2][0] + input[2][1]) / 4;
    weight_prime[1][1] = (input[1][1] + input[1][2] + input[2][1] + input[2][2]) / 4;
    bias_prime = 1
```

ج)

```
Forward(input, weight1, bias1, weight2, bias2, weight3, bias3, weight4, bias4):
    output[0][0] = weight1[0][0] * input[0][0] + weight1[0][1] * input[0][1] +
            weight1[1][0] * input[1][0] + weight1[1][1] * input[1][1] + bias1;
```

output[0][1] = weight2[0][0] * input[0][1] + weight2[0][1] * input[0][2] +
weight2[1][0] * input[1][1] + weight2[1][1] * input[1][2] + bias2;
output[1][0] = weight3[0][0] * input[1][0] + weight3[0][1] * input[1][1] +
weight3[1][0] * input[2][0] + weight3[1][1] * input[2][1] + bias3;

output[1][1] = weight4[0][0] * input[1][1] + weight4[0][1] * input[1][2] +
weight4[1][0] * input[2][1] + weight4[1][1] * input[2][2] + bias4;

Compute_ derivative:

Result = (weight1[0][0] * input[0][0] + weight1[0][1] * input[0][1] +
weight1[1][0] * input[1][0] + weight1[1][1] * input[1][1] +
weight2[0][0] * input[0][1] + weight2[0][1] * input[0][2] +
weight2[1][0] * input[1][1] + weight2[1][1] * input[1][2] +
weight3[0][0] * input[1][0] + weight3[0][1] * input[1][1] +
weight3[1][0] * input[2][0] + weight3[1][1] * input[2][1] +
weight4[0][0] * input[1][1] + weight4[0][1] * input[1][2] +
weight4[1][0] * input[2][1] + weight4[1][1] * input[2][2] +
bias1+bias2+bias3+bias4) / 4;

weight1_prime[0][0] = input[0][0] / 4;
weight1_prime[0][1] = input[0][1] / 4;
weight1_prime[1][0] = input[1][0] / 4;
weight1_prime[1][1] = input[1][1] / 4;
bias1_prime = 1 / 4

```
weight2_prime[0][0] = input[0][1]  / 4;
weight2_prime[0][1] = input[0][2] / 4;
weight2_prime[1][0] = input[1][1]]  / 4;
weight2_prime[1][1] = input[1][2]  / 4;
bias2_prime = 1 / 4

weight3_prime[0][0] = input[1][0] / 4;
weight3_prime[0][1] = input[1][1] / 4;
weight3_prime[1][0] = input[2][0]  / 4;
weight3_prime[1][1] = input[2][1]  / 4;
bias3_prime = 1 / 4

weight4_prime[0][0] = input[1][1]  / 4;
weight4_prime[0][1] = input[1][2] / 4;
weight4_prime[1][0] = input[2][1]]  / 4;
weight4_prime[1][1] = input[2][2]  / 4;
bias4_prime = 1 / 4
```

-4

دو مدل کد نوشته شده است که در یکی از MaxPoolling استفاده شده و در دیگری AveragePoolling. نتایج برای AveragePoolling با اختلاف ناچیزی بهتر است. در این مدل شبکه عصبی به ترتیب سه بار سه لایه‌ی کانولوشن قرار گرفته که تعداد 32 فیلتر دارد و اندازه‌ی کرنل (5,5) بوده و بعد از این سه لایه یک لایه‌ی Pooling قرار دارد. اندازه‌ی Pooling و stride هر دو (2,2) می‌باشد و تعداد پارامترهای شبکه حدودا 212هزارتا می‌باشد

تعداد داده‌ها کم اما تعداد کلاس‌ها 6 تاست. بنابیراین انتخاب 100 ایپاک برای train طبیعی است. دقت در ابتدا بسیار پایین است و تا حدود 20 ایپاک اول دقت روی داده‌ی validation از داده‌ی train بیشتر است که دلیل آن این است که augmentationهایی که روی داده‌ی train انجام دادیم مدل را سخت‌تر کرده است. و از اینجا به بعد دقت روی هر دو داده از 70 بیشتر می‌شود و حدود ایپاک 23، 24 دقت از 80 درصد بیشتر می‌شود. حدود ایپاک 50 دقت به 90 رسیده و loss در داده‌های train و validation از 2 به 0.2 رسیده یعنی یک دهم برابر شده است و تقریبا از همین جا به بعد loss در داده‌ی train بیشتر کاهش داشته اما در داده‌ی validation روند تقریبا ثابتی دارد و یا نرخ کاهش آن بسیار کند است. در ایپاک 77 دقت در داده‌ی validation برابر 98 درصد است و بالاترین دقت است و هنوز مدل overfit نشده است. در نهایت دقت در داده train به 97 درصد رسیده و در داده‌ی تست حدود 92 درصد است.