گزارش پروژه محاسبات علمی

نام و نام خانوادگی : صالح زارع زاده

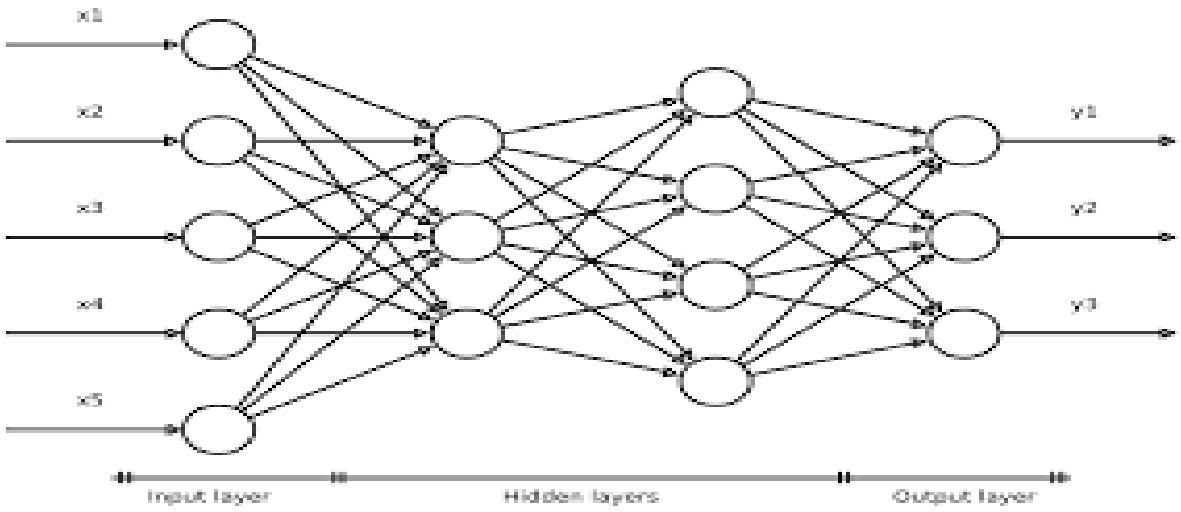
digit Recognition Using Convolutional Neural Network

یادگیری عمیق : که در زبان فارسی به یادگیری ژرف نیز ترجمه شده است، (Deep learning) یک زیر شاخه از یادگیری ماشین و بر مبنای مجموعه‌ای از الگوریتم‌ها است که در تلاشند تا مفاهیم انتزاعی سطح بالا در دادگان را مدل نمایند که این فرایند را با استفاده از یک گراف عمیق که دارای چندین لایه پردازشی متشکل از چندین لایه تبدیلات خطی و غیرخطی هستند، مدل می‌کنند.

یادگیری عمیق، رده ای از الگوریتم‌های یادگیری ماشین است که از چندین لایه برای استخراج ویژگی‌های سطح بالا از ورودی خام استفاده می‌کنند. به بیانی دیگر، رده‌ای از تکنیک‌های یادگیری ماشین که از چندین لایه‌ی پردازش اطلاعات و به ویژه اطلاعات غیرخطی بهره می‌برد تا عملیات تبدیل یا استخراج ویژگی نظارت‌شده یا نظارت‌نشده را عموماً با هدف تحلیل یا بازشناخت الگو، کلاس‌بندی، خوشه‌بندی انجام دهد.

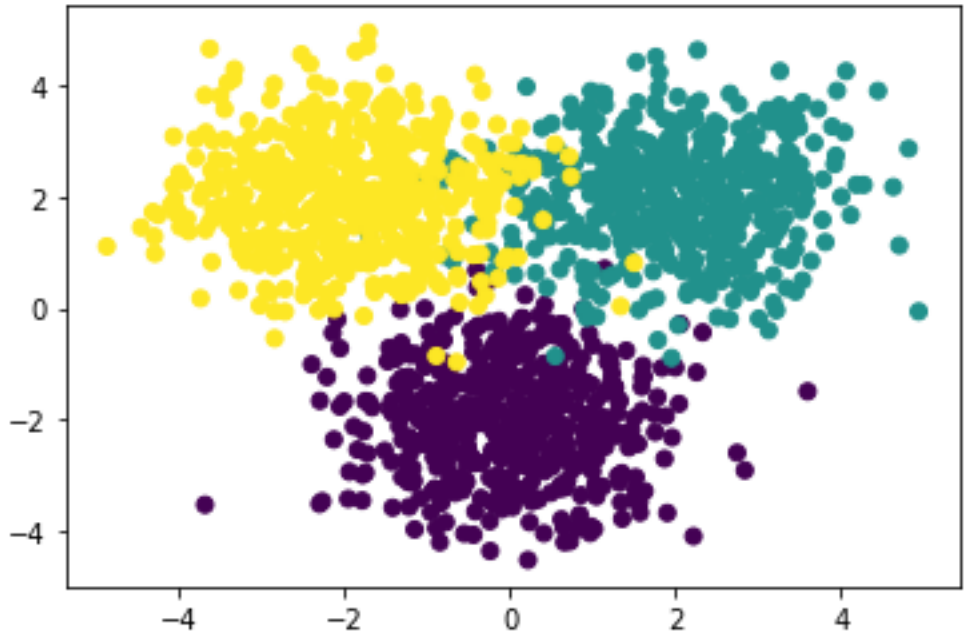
برای مثال، در پردازش تصویر، لایه‌های پست‌تر می‌توانند لبه‌ها را تشخیص دهند، در حالی که لایه‌های عالی‌تر ممکن است ویژگی‌های پرمعناتر برای انسان، همچون حروف یا چهره‌ها، را تشخیص دهند.

یادگیری عمیق زیرشاخه‌ای از یادگیری ماشین است که از لایه‌های متعدد تبدیلات خطی به منظور پردازش سیگنال‌های حسی مانند صدا و تصویر استفاده می‌کند. ماشین در این روش هر مفهوم پیچیده را به مفاهیم ساده‌تری تقسیم می‌کند، و با ادامه‌ی این روند به مفاهیم پایه‌ای می‌رسد که قادر به تصمیم‌گیری برای آن‌ها است و بدین ترتیب نیازی به نظارت کامل انسان برای مشخص کردن اطلاعات لازم ماشین در هر لحظه نیست. موضوعی که در یادگیری عمیق اهمیت زیادی دارد، نحوه‌ی ارائه‌ی اطلاعات است. ارائه دادن اطلاعات به ماشین باید به شیوه‌ای باشد که ماشین در کمترین زمان اطلاعات کلیدی را که می‌تواند با استناد به آن‌ها تصمیم بگیرد را دریافت کند.

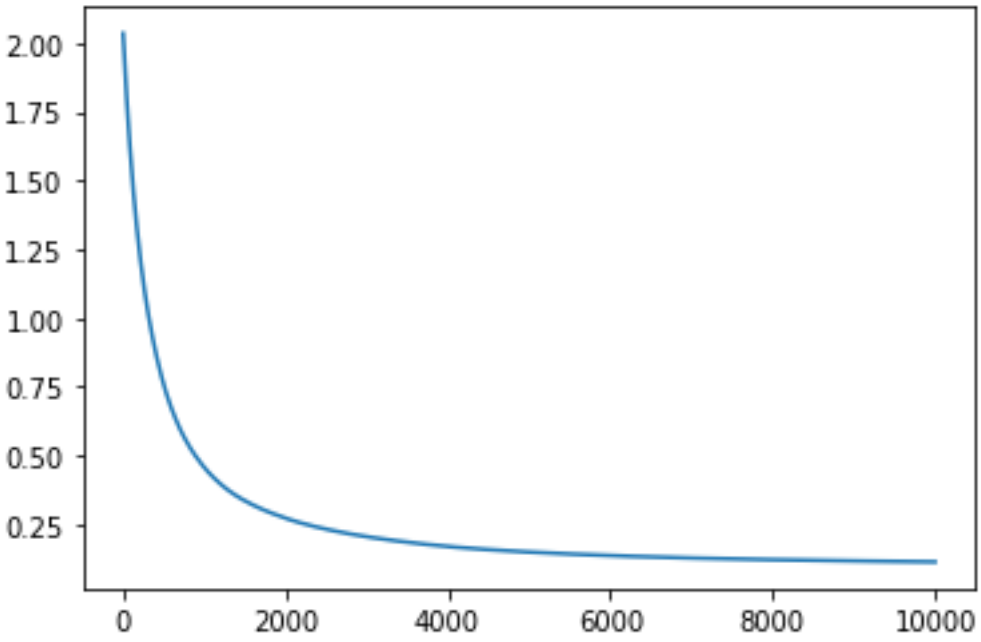


در واقع در این قسمت در theano_ann یک neural network طراحی شده است که داده های آن ۳ دایره با مرکز های متفاوت هستند و با تعداد iteration های مناسب میتواند به دقت خوبی برسد و این را میتوان برای هر داده ای استفاده کرد(تصویری هم میشود فقط باید قبل از آن تصویر از سه بعدی به صورت یک بعدی تبدیل شود در واقع ابتدا تصویر را به یک تصویر سیاه و سفید تبدیل میکنیم سپس ان تصویر دوبعدی را به صورت flat در میاوریم)

خروجی برای داده های زیر را نمایش میدهیم:



تابع هزینه :



خروجی بعد از هر ۵۰۰ ایتریشن برای ۱۰۰۰۰ ایتریشن:

i: 500 cost: 0.7580649139531994 score: 0.7153333333333334

i: 1000 cost: 0.4559896207495282 score: 0.8293333333333334

i: 1500 cost: 0.3347016341821057 score: 0.904

i: 2000 cost: 0.2714213539317616 score: 0.934

i: 2500 cost: 0.23125233691663713 score: 0.9433333333333334

i: 3000 cost: 0.20387897938258492 score: 0.9526666666666667

i: 3500 cost: 0.18377268749808237 score: 0.958

i: 4000 cost: 0.16823675641038582 score: 0.96

i: 4500 cost: 0.1567825506261278 score: 0.9606666666666667

i: 5000 cost: 0.14796165070012407 score: 0.9613333333333334

i: 5500 cost: 0.1409208439268261 score: 0.9613333333333334

i: 6000 cost: 0.13508721264210022 score: 0.9613333333333334

i: 6500 cost: 0.13019809705384838 score: 0.9626666666666667

i: 7000 cost: 0.12626429835283123 score: 0.9626666666666667

i: 7500 cost: 0.12297811229766094 score: 0.9633333333333334

i: 8000 cost: 0.12017259146374827 score: 0.962

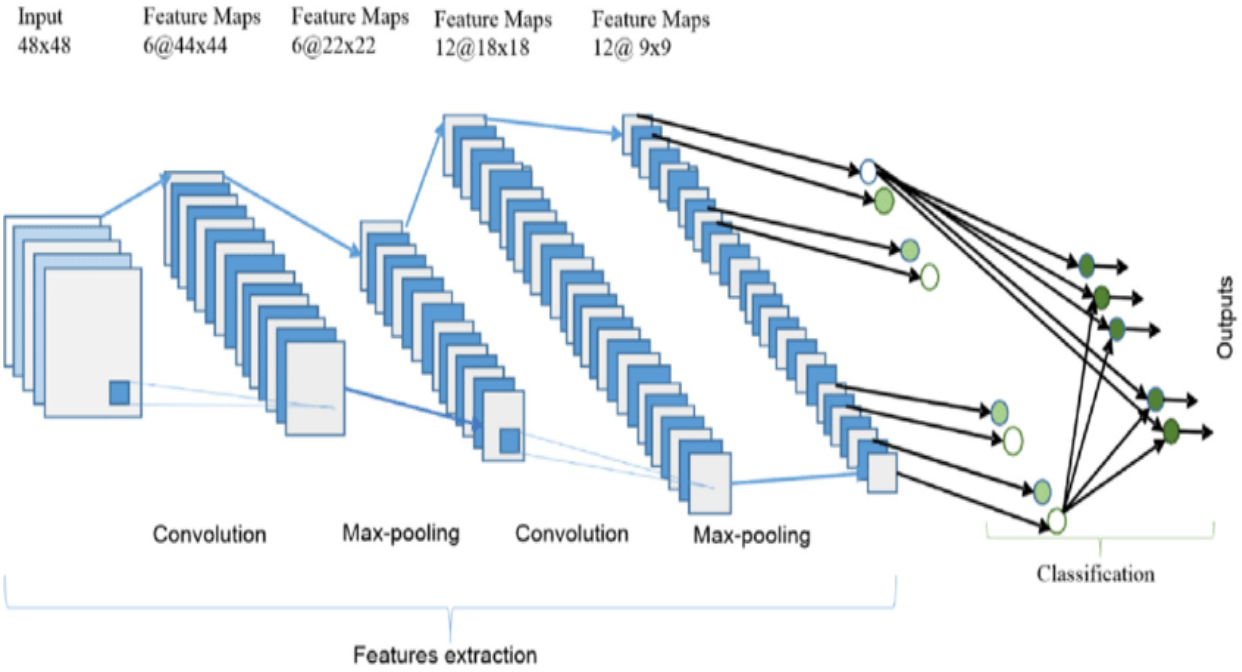
i: 8500 cost: 0.11777835935324081 score: 0.9626666666666667

i: 9000 cost: 0.11567380456477319 score: 0.9626666666666667

i: 9500 cost: 0.11384834563444575 score: 0.964

i: 10000 cost: 0.11222043196498148 score: 0.964

شبکه‌های عصبی پیچشی یا همگشتی (convolutional neural network)رده‌ای از شبکه‌های عصبی ژرف هستند که معمولاً برای انجام تحلیل‌های تصویری یا گفتاری در یادگیری ماشین استفاده می‌شوند. شکل زیر نشاندهنده روند آن است :

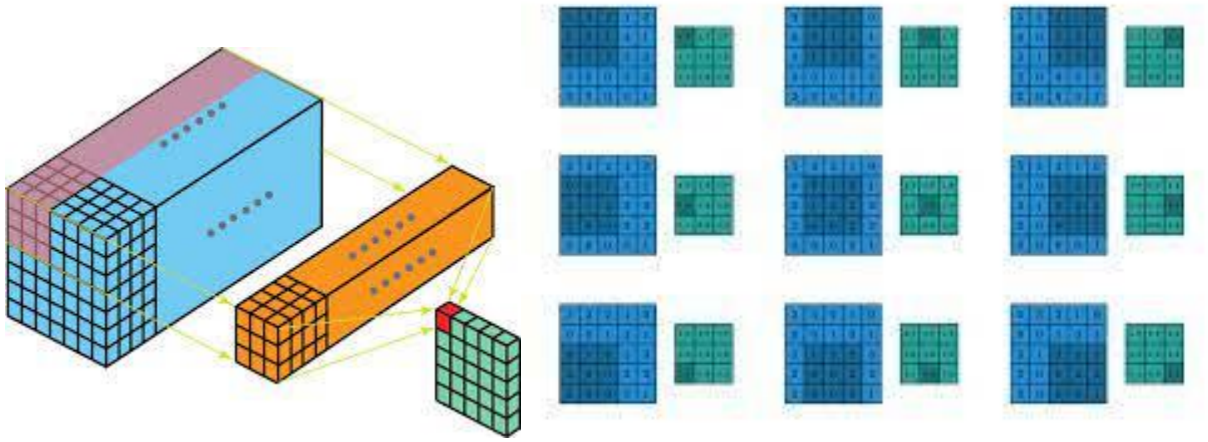


لایه‌های پیچشی

لایه‌های پیچشی یک عمل پیچش را روی ورودی اعمال می‌کنند، سپس نتیجه را به لایه بعدی می‌دهند. این پیچش در واقع پاسخ یک تک‌نورون را به یک تحریک دیداری شبیه‌سازی می‌کند.

هر نورون پیچشی داده‌ها را تنها برای ناحیه پذیرش خودش پردازش می‌کند. مشبک‌کردن به شبکه‌های پیچشی این اجازه را می‌دهد که انتقال، دوران یا با تبدیل ورودی داده ها را تصحیح کنند.

```
convpool_layer_sizes=[(20, 5, 5), (20, 5, 5)],
self.convpool_layers = []
        for mo, fw, fh in self.convpool_layer_sizes:
            layer = ConvPoolLayer(mi, mo, fw, fh)
            self.convpool_layers.append(layer)
            outw = (outw - fw + 1) / 2
            outh = (outh - fh + 1) / 2
            mi = mo
```

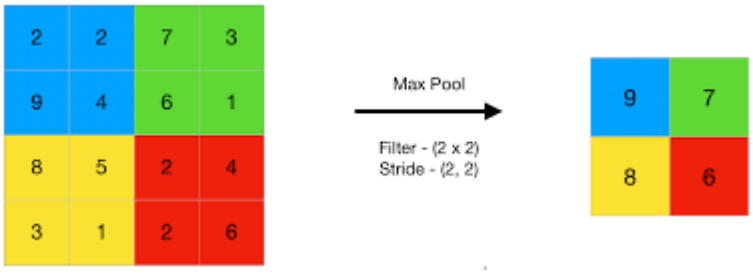


حال برای انجام این کار از این کد استفاده میکنیم:

```
conv_out = conv2d(input=X, filters=self.W)
```

لایه‌های ادغام

لایه‌های ادغام :(pooling layer) شبکه‌های عصبی پیچشی ممکن است شامل لایه‌های ادغام محلی یا سراسری باشند که خروجی‌های خوشه‌های نورونی در یک لایه را در یک تک‌نورون در لایه بعدی ترکیب می‌کند .به عنوان مثال روش حداکثر تجمع (max pooling) حداکثر مقدار بین خوشه‌های نورونی در لایه پیشین استفاده می‌کند مثال دیگر میانگین تجمع (average pooling) است که از مقدار میانگین خوشه‌های نورونی در لایه پیشین استفاده می‌کند.




```
pooled_out = pool.pool_2d(
            input=conv_out,
            ws=self.poolsz,
            ignore_border=True
        )
```

وزن‌ها

شبکه‌های عصبی پیچشی وزن‌ها را در لایه‌های پیچشی به اشتراک می‌گذارند که باعث می‌شود حداقل حافظه و بیشترین کارایی بدست بیاید.یادگیری ماشینی با نظارت (supervised learning) به دنبال تابعی از میان یک سری توابع هست که تابع هزینه(loss function) داده ها را بهینه سازد. به عنوان مثال در مسئله رگرسیون تابع هزینه می‌تواند اختلاف بین پیش‌بینی و مقدار واقعی خروجی به توان دو باشد، یا در مسئله طبقه بندی ضرر منفی لگاریتم احتمال خروجی باشد.
مشکلی که در یادگیری شبکه های عصبی وجود دارد این است که این مسئله بهینه سازی دیگر محدب (convex) نیست .ازین رو با مشکل کمینه‌های محلی یا local minimum روبرو هستیم. یکی از روش‌های متداول حل مسئله بهینه سازی در شبکه‌های عصبی بازگشت به عقب یا همان back propagation است . روش بازگشت به عقب گرادیانِ تابع هزینه را برای تمام وزن‌های شبکه عصبی محاسبه می‌کند و بعد از روش‌های گرادیان کاهشی (gradient descent) برای پیدا کردن مجموعه وزن‌های بهینه استفاده می‌کند . روش‌های گرادیان کاهشی سعی میکنند به صورت متناوب در خلاف جهت گرادیان حرکت کنند و با این کار تابع هزینه را به حداقل برسانند . پیدا کردن گرادیانِ لایه آخر ساده است و با استفاده از مشتق جزئی بدست می آید. گرادیانِ لایه‌های میانی اما به صورت مستقیم بدست نمی‌آید و باید از روش‌هایی مانند قاعده زنجیری در مشتق‌گیری استفاده کرد. روش بازگشت به عقب از قاعده زنجیری برای محاسبه گرادیان‌ها استفاده می‌کند و همانطور که در پایین خواهیم دید، این روش به صورت متناوب گرادیان‌ها را از بالاترین لایه شروع کرده آن‌ها را در لایه‌های پایینتر «پخش» می‌کند.

```
class HiddenLayer(object):
    def __init__(self, M1, M2, an_id):
        self.id = an_id
        self.M1 = M1
        self.M2 = M2
        W , b = init_weight_and_bias(M1, M2)
        self.W = theano.shared(W, 'W_%s' % self.id)
        self.b = theano.shared(b, 'b_%s' % self.id)
        self.params = [self.W, self.b]

    def forward(self, X):
        return T.nnet.relu(X.dot(self.W) + self.b)
```

بازگشت به عقب(Backpropagation) ، روشی برا محاسبه گرادیانها

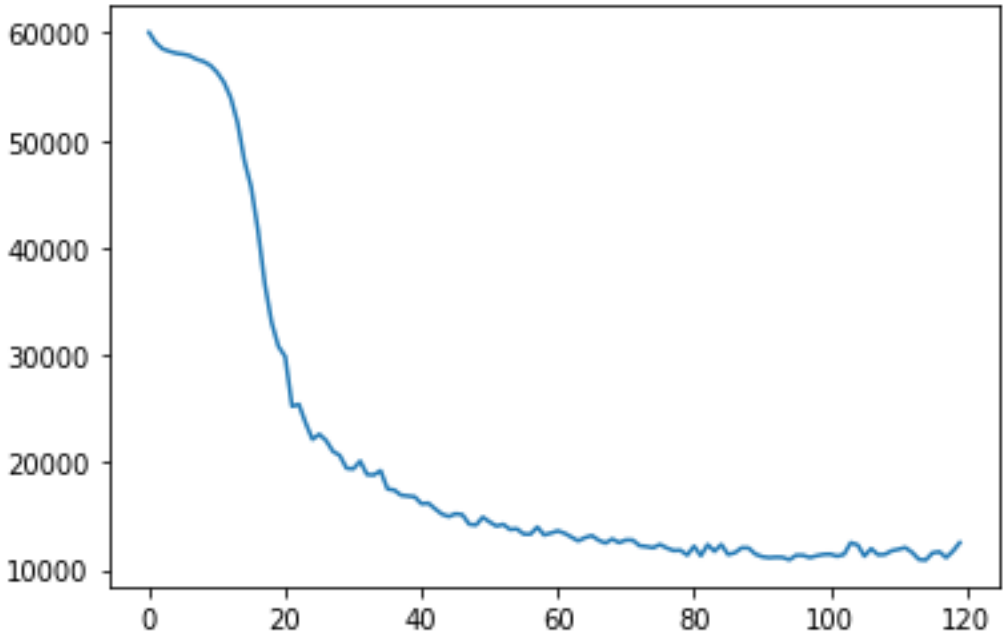
هدف کوچک کردن یک تابع هزینه بر روی تمام داده هاست با استفاده از گرادیان بازگشتی است که درواقع در هر مرحله مشتق تابع خطا به وزن های لایه ای که قرار است بهتر شود گرفته میشود (در در صورت نیاز regularization نیز به آن اضافه میشود) و با ضریبی که آن را learning rate نامیده میشود از وزن های آن لایه کم میشود و این روند برای لایه های قبلی ادامه پیدا میکند تا همه لایه ها نسبت به خطا بهینه تر شوند و سپس دوباره با استفاده از این لایه ها (ماتریس ها) مقادیر خروجی محاسبه شده و این کار تا زمانی که مقدار کاهش خطا کم شود یا به سمت overfitting بر روی داده ها برسیم ادامه پیدا میکند و سپس مدل به عنوان خروجی داده میشود.

```
K = len(set(Y))
        self.hidden_layers = []
        M1 = self.convpool_layer_sizes[-
1][0]*outw*outh # size must be same as output of last convpool layer
        count = 0
        for M2 in self.hidden_layer_sizes:
            h = HiddenLayer(M1, M2, count)
            self.hidden_layers.append(h)
            M1 = M2
            count += 1
W, b = init_weight_and_bias(M1, K)
        self.W = theano.shared(W, 'W_logreg')
        self.b = theano.shared(b, 'b_logreg')
```

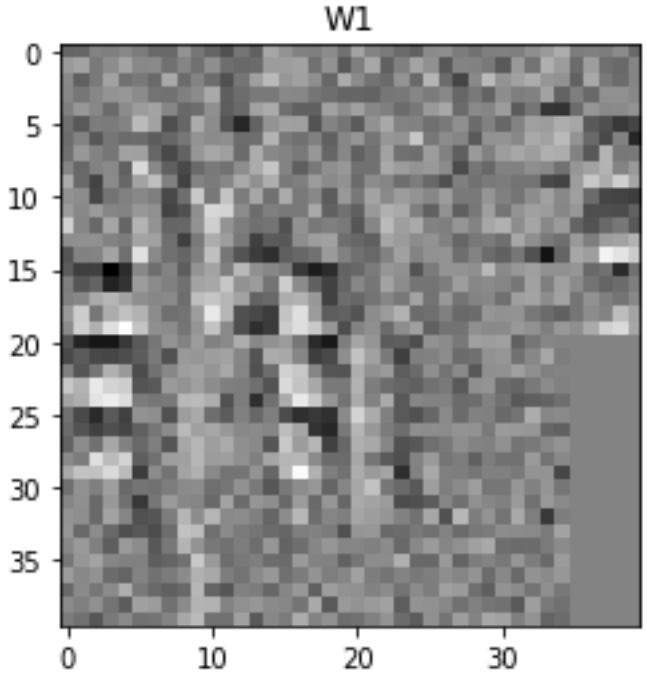
```
rcost = reg*T.sum([(p*p).sum() for p in self.params])
        cost = -T.mean(T.log(pY[T.arange(thY.shape[0]), thY])) + rcost
updates = [
            (p, p + mu*dp - lr*T.grad(cost, p)) for p, dp in zip(self.params, dpa
rams)
        ] + [
            (dp, mu*dp - lr*T.grad(cost, p)) for p, dp in zip(self.params, dparam
s)
        ]

        train_op = theano.function(
            inputs=[thX, thY],
            updates=updates
        )
```

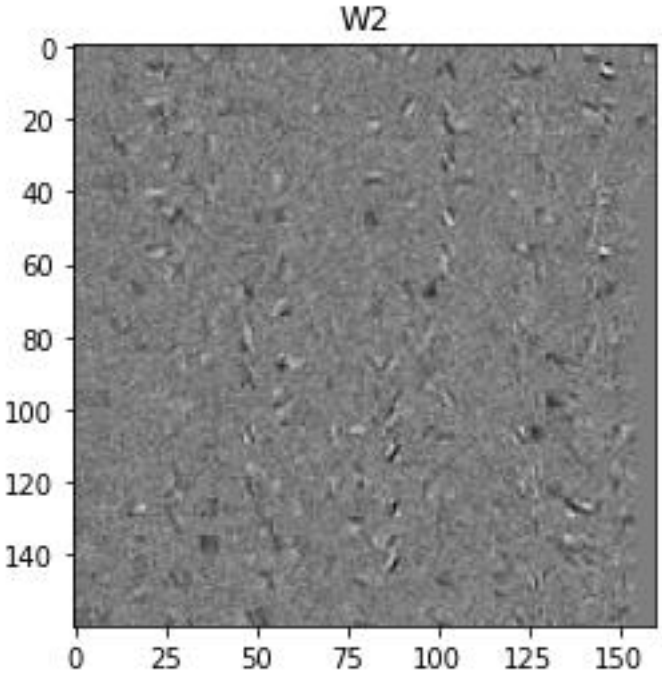
تابع هزینه برای داده های ورودی:



Convolution 1:



Convolution 2:



خروجی بعد از هر ۱۰ ایتریشن بر روی داده ها:

Cost / err at iteration i=0, j=0: 60050.138 / 0.897

Cost / err at iteration i=0, j=10: 59087.779 / 0.816

Cost / err at iteration i=0, j=20: 58515.661 / 0.804

Cost / err at iteration i=0, j=30: 58284.759 / 0.804

Cost / err at iteration i=0, j=40: 58104.514 / 0.804

Cost / err at iteration i=0, j=50: 58017.767 / 0.790

Cost / err at iteration i=0, j=60: 57867.144 / 0.747

Cost / err at iteration i=0, j=70: 57542.064 / 0.804

Cost / err at iteration i=0, j=80: 57340.380 / 0.804

Cost / err at iteration i=0, j=90: 56980.620 / 0.798

Cost / err at iteration i=0, j=100: 56328.592 / 0.753

Cost / err at iteration i=0, j=110: 55413.349 / 0.748

Cost / err at iteration i=0, j=120: 53976.684 / 0.701

Cost / err at iteration i=0, j=130: 51663.313 / 0.654

Cost / err at iteration i=0, j=140: 48123.898 / 0.604

Cost / err at iteration i=1, j=0: 45654.278 / 0.559

Cost / err at iteration i=1, j=10: 41631.275 / 0.526

Cost / err at iteration i=1, j=20: 36644.723 / 0.442

Cost / err at iteration i=1, j=30: 33040.571 / 0.399

Cost / err at iteration i=1, j=40: 30805.722 / 0.341

Cost / err at iteration i=1, j=50: 29848.401 / 0.331

Cost / err at iteration i=1, j=60: 25223.149 / 0.266

Cost / err at iteration i=1, j=70: 25407.412 / 0.267

Cost / err at iteration i=1, j=80: 23658.220 / 0.250

Cost / err at iteration i=1, j=90: 22144.867 / 0.235

Cost / err at iteration i=1, j=100: 22614.847 / 0.230

Cost / err at iteration i=1, j=110: 22035.207 / 0.224

Cost / err at iteration i=1, j=120: 21028.748 / 0.222

Cost / err at iteration i=1, j=130: 20644.593 / 0.219

Cost / err at iteration i=1, j=140: 19449.966 / 0.205

Cost / err at iteration i=2, j=0: 19396.696 / 0.205

Cost / err at iteration i=2, j=10: 20117.988 / 0.212

Cost / err at iteration i=2, j=20: 18866.708 / 0.198

Cost / err at iteration i=2, j=30: 18824.132 / 0.197

Cost / err at iteration i=2, j=40: 19220.507 / 0.204

Cost / err at iteration i=2, j=50: 17525.859 / 0.181

Cost / err at iteration i=2, j=60: 17421.071 / 0.182

Cost / err at iteration i=2, j=70: 16939.658 / 0.181

Cost / err at iteration i=2, j=80: 16867.608 / 0.180

Cost / err at iteration i=2, j=90: 16792.051 / 0.177

Cost / err at iteration i=2, j=100: 16142.107 / 0.168

Cost / err at iteration i=2, j=110: 16199.968 / 0.174

Cost / err at iteration i=2, j=120: 15671.914 / 0.170

Cost / err at iteration i=2, j=130: 15181.713 / 0.162

Cost / err at iteration i=2, j=140: 14955.080 / 0.160

Cost / err at iteration i=3, j=0: 15212.572 / 0.162

Cost / err at iteration i=3, j=10: 15123.801 / 0.163

Cost / err at iteration i=3, j=20: 14240.292 / 0.149

Cost / err at iteration i=3, j=30: 14162.198 / 0.149

Cost / err at iteration i=3, j=40: 14897.312 / 0.162

Cost / err at iteration i=3, j=50: 14447.672 / 0.154

Cost / err at iteration i=3, j=60: 14064.599 / 0.149

Cost / err at iteration i=3, j=70: 14227.633 / 0.153

Cost / err at iteration i=3, j=80: 13749.516 / 0.148

Cost / err at iteration i=3, j=90: 13808.089 / 0.149

Cost / err at iteration i=3, j=100: 13321.449 / 0.146

Cost / err at iteration i=3, j=110: 13315.365 / 0.145

Cost / err at iteration i=3, j=120: 13976.408 / 0.151

Cost / err at iteration i=3, j=130: 13237.838 / 0.143

Cost / err at iteration i=3, j=140: 13412.931 / 0.145

Cost / err at iteration i=4, j=0: 13620.652 / 0.148

Cost / err at iteration i=4, j=10: 13392.410 / 0.146

Cost / err at iteration i=4, j=20: 13040.160 / 0.143

Cost / err at iteration i=4, j=30: 12684.035 / 0.137

Cost / err at iteration i=4, j=40: 12988.162 / 0.142

Cost / err at iteration i=4, j=50: 13179.018 / 0.141

Cost / err at iteration i=4, j=60: 12746.726 / 0.138

Cost / err at iteration i=4, j=70: 12477.728 / 0.138

Cost / err at iteration i=4, j=80: 12865.146 / 0.139

Cost / err at iteration i=4, j=90: 12517.686 / 0.136

Cost / err at iteration i=4, j=100: 12771.640 / 0.138

Cost / err at iteration i=4, j=110: 12757.862 / 0.140

Cost / err at iteration i=4, j=120: 12234.029 / 0.133

Cost / err at iteration i=4, j=130: 12157.955 / 0.132

Cost / err at iteration i=4, j=140: 12049.453 / 0.128

Cost / err at iteration i=5, j=0: 12350.853 / 0.129

Cost / err at iteration i=5, j=10: 12025.048 / 0.126

Cost / err at iteration i=5, j=20: 11782.007 / 0.128

Cost / err at iteration i=5, j=30: 11816.541 / 0.127

Cost / err at iteration i=5, j=40: 11357.151 / 0.120

Cost / err at iteration i=5, j=50: 12192.298 / 0.128

Cost / err at iteration i=5, j=60: 11305.216 / 0.121

Cost / err at iteration i=5, j=70: 12339.754 / 0.134

Cost / err at iteration i=5, j=80: 11710.727 / 0.124

Cost / err at iteration i=5, j=90: 12345.757 / 0.135

Cost / err at iteration i=5, j=100: 11413.855 / 0.119

Cost / err at iteration i=5, j=110: 11584.681 / 0.123

Cost / err at iteration i=5, j=120: 12042.725 / 0.128

Cost / err at iteration i=5, j=130: 12028.677 / 0.129

Cost / err at iteration i=5, j=140: 11455.979 / 0.121

Cost / err at iteration i=6, j=0: 11212.295 / 0.119

Cost / err at iteration i=6, j=10: 11124.498 / 0.116

Cost / err at iteration i=6, j=20: 11168.470 / 0.118

Cost / err at iteration i=6, j=30: 11149.374 / 0.117

Cost / err at iteration i=6, j=40: 10928.701 / 0.116

Cost / err at iteration i=6, j=50: 11350.193 / 0.118

Cost / err at iteration i=6, j=60: 11341.667 / 0.120

Cost / err at iteration i=6, j=70: 11134.606 / 0.119

Cost / err at iteration i=6, j=80: 11302.569 / 0.119

Cost / err at iteration i=6, j=90: 11427.228 / 0.121

Cost / err at iteration i=6, j=100: 11448.884 / 0.119

Cost / err at iteration i=6, j=110: 11262.064 / 0.116

Cost / err at iteration i=6, j=120: 11428.561 / 0.120

Cost / err at iteration i=6, j=130: 12509.636 / 0.125

Cost / err at iteration i=6, j=140: 12316.311 / 0.129

Cost / err at iteration i=7, j=0: 11267.232 / 0.116

Cost / err at iteration i=7, j=10: 11960.868 / 0.125

Cost / err at iteration i=7, j=20: 11388.564 / 0.118

Cost / err at iteration i=7, j=30: 11424.488 / 0.119

Cost / err at iteration i=7, j=40: 11763.965 / 0.121

Cost / err at iteration i=7, j=50: 11898.589 / 0.117

Cost / err at iteration i=7, j=60: 12068.036 / 0.118

Cost / err at iteration i=7, j=70: 11613.937 / 0.120

Cost / err at iteration i=7, j=80: 10956.606 / 0.111

Cost / err at iteration i=7, j=90: 10904.979 / 0.115

Cost / err at iteration i=7, j=100: 11568.481 / 0.120

Cost / err at iteration i=7, j=110: 11674.262 / 0.116

Cost / err at iteration i=7, j=120: 11131.873 / 0.115

Cost / err at iteration i=7, j=130: 11736.624 / 0.118

Cost / err at iteration i=7, j=140: 12524.691 / 0.124

Elapsed time: 0:45:43.598494