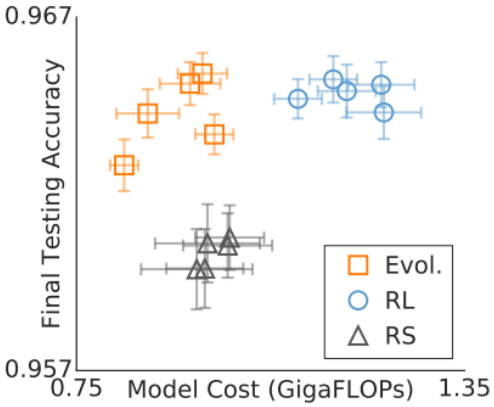
- هدف پروژه : در این پروژه به کمک الگوریتم ارائه‌شده در مقاله‌ی Regularized Evolution for Image Classifier Architecture Search، قصد داریم تا به کمک الگوریتم ژنتیک، یک ساختار بهینه برای کلاس‌بندی عکس یک دیتاست (مثلا cifar10) بیابیم. (همانطور که می‌دانید در ابتدا قرار بود که از داده‌های ورودی حسگرهای یک ربات برای این کار استفاده کنیم، ولی به دلیل کمبود بعد این داده‌ها (فقط ۲۴ بعد) از داده‌هایی با بعد بیشتر(32x32 بعد) مانند عکس استفاده کردیم. در ابتدا به بررسی توابع پیاده‌سازی شده در کد و شرح جزئیات الگوریتم خواهیم پرداخت.



شکل مقابل، روش‌های معروف به کارگیری شده برای عملیات NAS را نشان می‌دهد. همان‌گونه که دیده می‌شود، RS (Random Search) به دلیل عدم وجود استراتژی برای جستوجوی محیط، به تناسب خوبی برای خروجی دست نمی‌یابد، ولی چون هیچ قیدی بر روی زمان ندارد (عملا شرطی برای پایان زمان الگوریتم نمی‌توان یافت بنابراین تعداد بررسی‌ها ثابت فرض می‌شود) به مرتبه‌ی زمانی خوبی دست یافته. در سر دیگر طیف، الگوریتم‌های RL قرار دارند که با به کارگیری یک ساختارRNN و استفاده از یادگیری تقویتی برای آموزش آن، ساختار شبکه را می‌یابند، همانگونه که در شکل دیده می‌شود، این الگوریتم‌ها به نتایج عالی دست یافته اند اما هزینه‌ی فراوان دارند. رویکرد سوم اما، بهره‌گیری از الگوریتمی است که علاوه بر سرعت خوب در جستوجو، دارای استراتژی (هر چند ناچیز) برای جستوجو باشد. الگوریتم‌های تکاملی کاندیدای خوبی برای این نوع الگوریتم می‌باشند. در پروژه‌ی پایانی این الگوریتم را برای شبکه‌های CNN پیاده‌سازی شده است. تمامی تابع‌های نوشته شده تست شده و خروجی آن‌ها در گزارش آورده شده است. به دلیل زمان‌بر بودن شدید الگوریتم، از آزمایش آن پرهیز کرده ایم (در مقاله‌ی اصلا از ۴۵۰ GPU در مدت ۷ روز استفاده شده است و همانگونه که در ادامه می‌بینیم، ساختارهای شبکه‌های پیشنهاد شده دارای پارامترهای زیادی است _ شبکه‌ی استفاده شده در مقاله دارای ۳۳میلیون پارامتر است_ که حتی اصلاح شبکه و ساده‌سازی آن نیز کمک زیادی به کم شدن آن و قابل اجرا شدن شبیه‌سازی نمی‌کند.)

در ادامه به بررسی توابع پیاده‌سازی شده و جزئیات مقاله می‌پردازیم :

لایه‌های مجاز در شبکه :

```python
def operation(arch):
    # arch = (I,F,T)
    if arch[2]==0:
        rc = random.randint(0, 2)
        if rc == 0:
            return nn.Conv2d(arch[0],arch[1],kernel_size=3,stride=1,padding=1)
        if rc == 1:
            return nn.Conv2d(arch[0],arch[1],kernel_size=5,stride=1,padding=2)
        if rc == 2:
            return nn.Conv2d(arch[0],arch[1],kernel_size=7,stride=1,padding=3)
    if arch[2] in [x+1 for x in range(5)]: return nn.AvgPool2d(kernel_size=3,stride=1,padding=1)
    if arch[2] in [x+5 for x in range(5)]: return nn.MaxPool2d(kernel_size=3,stride=1,padding=1)
    if arch[2] in [x+9 for x in range(5)]: return nn.Identity()
    if arch[2] in [x+13 for x in range(5)]: return nn.BatchNorm2d(num_features=arch[0])
    if arch[2]==18: return nn.MaxPool2d(kernel_size=3,stride=2,padding=1)
```

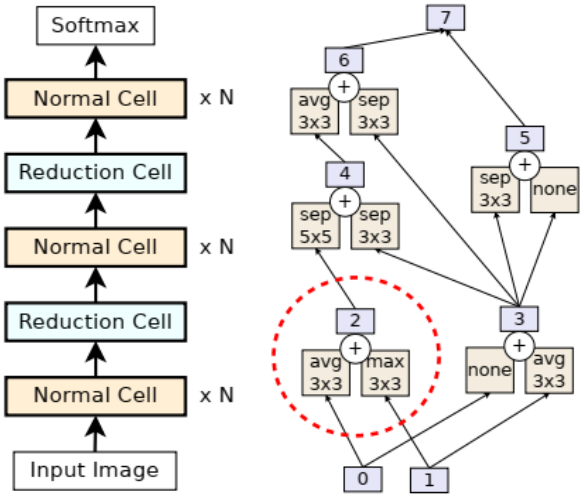
طبق شکل بالا و مقاله، استفاده از لایه‌های کانوولوشن در ابعاد مختلف که اندازه‌ی ورودی را تغییر نمی‌دهند، لایه میانگین و بیشینه‌گیر و همچنین لایه‌ی بی‌اثر در ساخت شبکه مجاز است. لایه‌ی آخر با طول قدم ۲، تنها در سلول‌های کاهنده مورد استفاده قرار می‌گیرد و مساحت عکس را ¼ عکس اصلا می‌نماید. در ابتدا به بررسی سلول‌ها و محتوای آن‌های می‌پردازیم.

ترکیب‌های دوتایی :

همانگونه که در ادامه توضیح خواهیم داد، هر سلول شبکه‌ی CNN، دارای تعداد حالت می‌باشد. این حالت‌ها به کمک این ترکیب های دوتایی پیاده‌سازی شده اند: به عبارت دقیق تر، به جز حالت اول و دوم که ورودی سلول محسوب می‌شود، تمامی حالت‌های دیگر خروجی‌های ترکیب‌های دوتایی هستند.

```python
class PairwiseCombination(nn.Module):
    def __init__(self,ARCH=[]):
        # ARCH = [arch1=(IC1,FC1,T1),arch2=(IC2,FC2,T2)]
        super(PairwiseCombination, self).__init__()
        self.op1 = operation(ARCH[0])
        self.op2 = operation(ARCH[1])

    def forward(self, x1, x2):
        x1 = F.relu(self.op1(x1))
        x2 = F.relu(self.op2(x2))
        return torch.cat(tensors=(x1,x2),dim=1)
```

در شکل راست زیر، محتوای یک سلول نمایش داده شده است، که دارای ۷ حالت و ۵ ترکیب دوتایی مختلف با فیلترهای مجاز می‌باشد. شکل سمت چپ، شمای کلی شبکه را نشان می‌دهد، همانگونه که دیده می‌شود، هر شبکه دارای ۵ استک از سلول‌های نرمال و کاهنده است که در انتها به کمک یک کلاس‌بند (در اینجا یک شبکه‌ی کاملا متصل) و یک لایه‌ی softmax، عمل کلاس بندی را انجام خواهد داد.



ساختار سلول‌ها :

همانگونه که در بالا هم توضیح داده شد، سلول‌ها دارای چندین حالت هستند که به صورت یک گراف جهت‌دار بدون دور، به همدیگر ربط دارند، هر حالت دقیقا ۲ ورودی دارد و می‌تواند یک یا بیشتر خروجی داشته باشد. چالش ساختن سلول‌ها، ساخت آن‌ها به شیوه‌ی پویا است. در ادامه با متدی که این کار را انجام می‌دهد آشنا خواهیم شد.

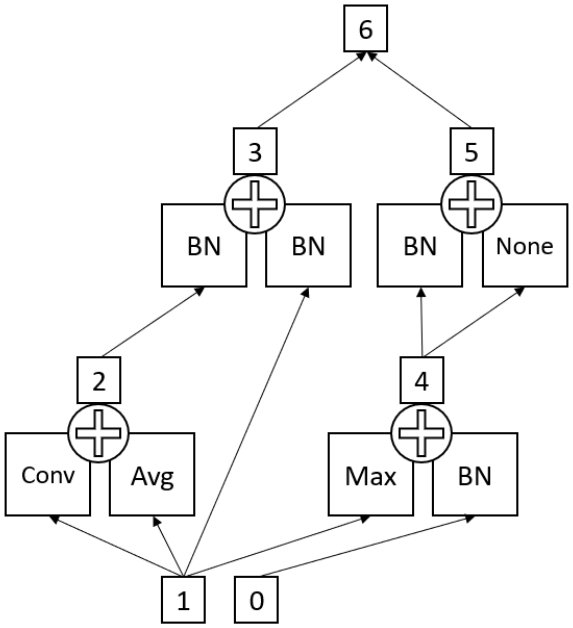
```python
class Cell(nn.Module):
    def __init__(self, ps_ARCH_list, dag_list, reduce=False):
        super(Cell, self).__init__()
        self.ps_list = [PairwiseCombination(ARCH) for ARCH in ps_ARCH_list[2:]]
        self.reduce = reduce
        self.dag_list = dag_list
        self.ps_ARCH_list = ps_ARCH_list

    # ps_list = [-1,-1, ps_net1, ps_net2,...]
    def forward(self, x):
        s = [x,x]
        for into,frm in enumerate(self.dag_list[2:-1]):
            s += [ self.ps_list[into].forward(s[frm[0]], s[frm[1]]) ]
        # concatinate result
        s += [ torch.cat(tensors=[s[state] for state in self.dag_list[-1]],dim=1) ]
        if self.reduce: return operation(arch=[-1,-1,MAX_PS_TYPE+1]).forward(s[-1])
        else: return s[-1]
```

ساخت پویای سلول‌های شبکه :

یکی از مهم‌ترین تابع‌های پروژه که عمل ساخت و تغییر سلول‌های شبکه را به صورت پویا انجام می‌دهد، تابع مقابل است. به صورت کلی هر سلول به صورت کامل به وسیله‌ی گراف جهت‌دار بدون دور و با تعیین تمامی حالت هایش (جفت‌های ترکیبی) مشخص می‌شود. تنها مشکل تعیین ابعاد ورودی و خروجی این سلول‌ها است که بایستی آن‌ها را به تدریج و با عبوردهی ورودی فرضی در سلول (یک نمونه از دیتاست برای داشتن سایز) تعیین کرده و سلول را تدریجا بسازیم. این تابع می‌تواند سلول را از پایه به صورت تصادفی بسازد و یا اینکه گراف ویا حالت آن را تغییر دهد. در رابطه با این دو ویژگی در تابع جهش بخش تکامل بحث خواهیم کرد. در ادامه شکل یک سلول تصادفی تولید شده از تابع را می‌بینیم :

```python
def generate_cell(f,x0, reduce=False,log=False,random_dag_list=True,random_ps_list=True,dag_list=[],arch_list=[]):
    # random number of states
    num_of_states = random.randint(MIN_NUM_OF_STATES, MAX_NUM_OF_STATES)
    # random DAG
    if random_dag_list:
        DAG = [-1,-1] + [[random.randint(0, 1+max_state),random.randint(0, 1+max_state)] for max_state in range(num_of_states)]
        # final node
        final_nodes = list(range(num_of_states+2))[2:]
        if log : print(final_nodes)
        for s_pair in DAG[2:]: final_nodes = list(set(final_nodes) - set(s_pair))
        DAG += [final_nodes]
    else : DAG = dag_list
    if log : print(DAG)
    # initiate ps list
    ps_list = [-1,-1]
    # initiate arch list
    ARCH_list = [-1,-1]
    # initiate input candidate list
    input_candidate_list = [x0, x0]
    # iterate through DAG and build
    for index,ps in enumerate(DAG[2:-1]):
        # update IC1, IC2
        IC1, IC2 = (input_candidate_list[ps[0]].shape)[1],(input_candidate_list[ps[1]].shape)[1]
        if random_ps_list:
            # make two random ps
            T1,T2 = [random.randint(0, MAX_PS_TYPE),random.randint(0, MAX_PS_TYPE)]
        else :
            T1,T2 = arch_list[index+2][0][2],arch_list[index+2][1][2]
        if log : print('ps={}, IC1={}, IC2={}, T_1={}, T_2={}'.format(ps,IC1,IC2,T1,T2))
        # make a random ps
        random_ps = PairwiseCombination(ARCH=[(IC1,f*IC1,T1),(IC2,f*IC2,T2)])
        # update ARCH list
        ARCH_list += [[(IC1,f*IC1,T1),(IC2,f*IC2,T2)]]
        # add ps to ps_list
        ps_list += [random_ps]
        # update ouput dimention
        input_candidate_list += [random_ps.forward(input_candidate_list[ps[0]],input_candidate_list[ps[1]])]
    if log : print('----------------------------end-of-cell-log-----')
    return Cell(ps_ARCH_list=ARCH_list,dag_list=DAG,reduce=reduce)
```

```python
nc1 = generate_cell(f=2,x0=x1, reduce=False,log=True,random_dag_list=True,random_ps_list=True,dag_list=[],arch_list=[])
y1 = nc1.forward(x1)
```



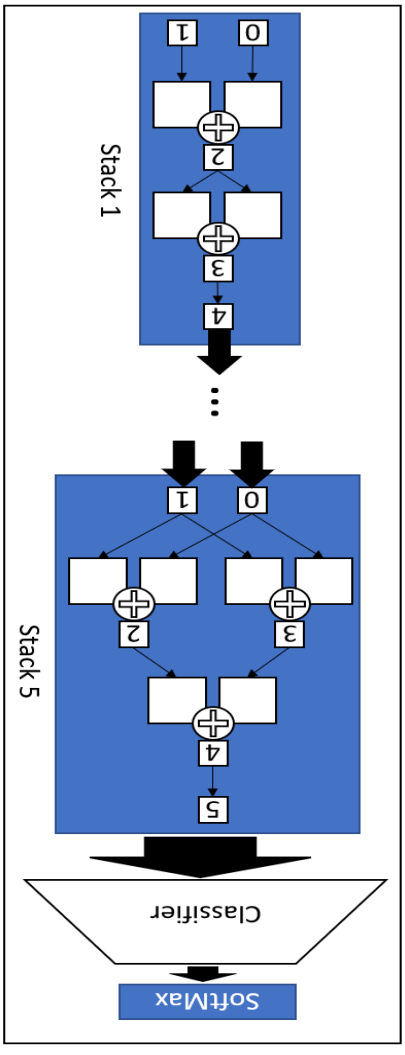

```
[2, 3, 4, 5]
[-1, -1, [1, 1], [1, 2], [0, 1], [4, 4], [3, 5]]
ps=[1, 1], IC1=1, IC2=1, T_1=0, T_2=1
ps=[1, 2], IC1=1, IC2=3, T_1=15, T_2=5
ps=[0, 1], IC1=1, IC2=1, T_1=17, T_2=17
ps=[4, 4], IC1=2, IC2=2, T_1=13, T_2=16
----------------------------end-of-cell-log-----
torch.Size([1, 1, 28, 28])
torch.Size([1, 8, 28, 28])
```

همانگونه که دیده می‌شود، لایه‌ی کانولوشن سبب شده است که تعداد کانال‌های ورودی از ۱ به ۸ افزایش بیابد. نمونه‌ی ترسیم شده تنها یک سلول است. شبکه‌دارای ۵ استک از انواع مختلف و تصادفی سلول های مختلف است. بنابراین همانگونه که انتظار داریم، بعد کانال‌های ورودی به شدت افزایش پیدا خواهد کرد. در ادامه این موضوع را در شبکه‌های کاندید مشاهده خواهید کرد.

شبکه‌های nasnet :

طبق محتوای مقاله، هر شبکه‌ی پیشنهادی دارای ۵ استک از سلول‌های نرمال و کاهنده است که هیچ رابطه‌ای با هم دیگر ندارند و کاملا به صورت تصادفی ساخته خواهند شد. هر شبکه دارای دو پارامتر N و F می‌باشد، که اولی تعداد سلول‌ها را در هر استک مشخص می‌کند، و پارامتر دوم نیز ضریب تعداد خروجی های لایه‌ی کانوولوشن است، برای مثال N=2,F=3 شبکه‌ای را توصیف می‌کند که دارای ۵ استک از سلول‌ها می‌باشد که هر استک ۲ سلول دارد، بنابراین شبکه دارای ۱۰ سلول تصادفی خواهد بود. همچنین F=3 بیان می‌کند که در صورت وجود لایه‌ی کانولوشن، تعداد ۳ فیلتر مختلف برای آن در نظر گرفته و آموزش داده شود. در ادامه یک شبکه‌ی ساخته شده به صورت تصادفی را می‌بینیم.

```
[2, 3]
[-1, -1, [0, 1], [2, 2], [3]]
ps=[0, 1], IC1=1, IC2=1, T_1=1, T_2=0
ps=[2, 2], IC1=2, IC2=2, T_1=16, T_2=16
-----------------------------end-of-cell-log-----
[2, 3]
[-1, -1, [0, 1], [0, 2], [3]]
ps=[0, 1], IC1=4, IC2=4, T_1=13, T_2=3
ps=[0, 2], IC1=4, IC2=8, T_1=15, T_2=6
-----------------------------end-of-cell-log-----
[2, 3, 4]
[-1, -1, [1, 1], [1, 0], [3, 3], [2, 4]]
ps=[1, 1], IC1=12, IC2=12, T_1=11, T_2=17
ps=[1, 0], IC1=12, IC2=12, T_1=12, T_2=16
ps=[3, 3], IC1=24, IC2=24, T_1=15, T_2=6
-----------------------------end-of-cell-log-----
[2, 3, 4]
[-1, -1, [1, 0], [1, 2], [2, 3], [4]]
ps=[1, 0], IC1=72, IC2=72, T_1=2, T_2=8
ps=[1, 2], IC1=72, IC2=144, T_1=15, T_2=13
ps=[2, 3], IC1=144, IC2=216, T_1=2, T_2=3
-----------------------------end-of-cell-log-----
[2, 3, 4]
[-1, -1, [0, 1], [0, 1], [3, 2], [4]]
ps=[0, 1], IC1=360, IC2=360, T_1=11, T_2=1
ps=[0, 1], IC1=360, IC2=360, T_1=6, T_2=1
ps=[3, 2], IC1=720, IC2=720, T_1=8, T_2=17
-----------------------------end-of-cell-log-----
[torch.Size([1, 1440, 7, 7])]
torch.Size([1, 1440, 7, 7])
torch.Size([1, 70560])
torch.Size([1, 10])
```

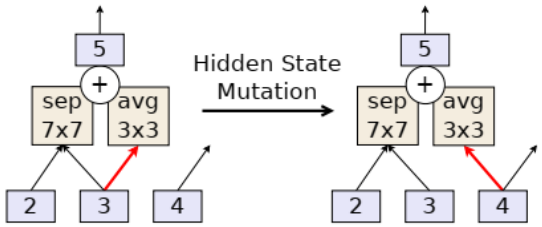


ساختار بالا، یک شبکه‌ی کامل است که قابلیت جهش و ارزیابی شدن را دارد. بنابراین می‌توان از این شبکه‌ها در الگوریتم ژنتیک استفاده کرد. در ادامه به تابع‌های جهش و برازش می‌پردازیم. توجه شود که طبق بیان مقاله، استفاده از عملگر ترکیب تاثیر زیادی در بهبود الگوریتم ندارد.

جهش نوع اول μ1 :

```python
def μ1(self, nasnet):
    # perfroms mutation by changing one random cell's DAG structure
    layer = nasnet.layers[random.randint(0, len(nasnet.layers)-1)]
    cell = layer[random.randint(0, len(layer)-1)]
    DAG = copy.deepcopy(cell.dag_list)
    psI = random.randint(2, len([-1,-1]+cell.ps_list)-2)
    DAG[psI][random.randint(0,1)] = random.randint(0,psI-1)
    return generate_cell(self.f,self.x0, reduce=cell.reduce,log=False,random_dag_list=False,random_ps_list=False,dag_list=DAG,arch_list=cell.ps_ARCH_list)
```

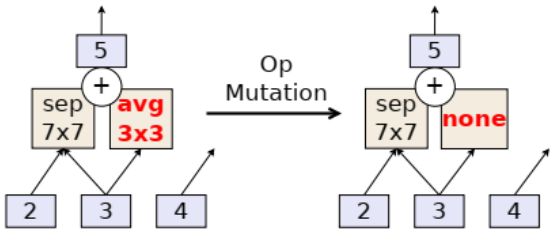
در این نوع جهش، ساختار گراف جهتدار بدون دور را تغییر می‌دهیم، به عبارتی دیگر، با انتخاب یکی از جفت‌های ترکیبی و تعویض یکی از ورودی‌های آن به صورت تصادفی، گراف را تغییر می‌دهیم.



جهش نوع دوم μ2 :

```python
def μ2(self, nasnet):
    # performs mutation by changing one random cell's PS operator
    layer = nasnet.layers[random.randint(0, len(nasnet.layers)-1)]
    cell = layer[random.randint(0, len(layer)-1)]
    ARCH_LIST = copy.deepcopy(cell.ps_ARCH_list)
    ARCH_LIST[random.randint(2, len(cell.ps_ARCH_list)-1)][random.randint(0, 1)][3] = random.randint(0, MAX_PS_TYPE)
    return generate_cell(self.f,self.x0, reduce=cell.reduce,log=False,random_dag_list=False,random_ps_list=False,dag_list=cell.dag_list,arch_list=ARCH_LIST)
```

در این نوع جهش، با انتخاب یک سلول به تصادف از شبکه و سپس انتخاب یک جفت ترکیبی از سلول، یکی از دو لایه‌ی آن را به صورت تصادفی تغییر می‌دهیم.



تابع برازش ϕ :

```python
def ϕ(self, nasnet):
    # measures fitness based on test_accuracy and resource use
    # first train the network
    train_network(nasnet, self.trainloader)
    # then test the network
    # add other fitness parameters if required
    return test_network(nasnet, self.testloader)
```

این تابع با دریافت شبکه، آن را آموزش می‌دهد و سپس میزان دقت آن را بر روی داده‌های تست اندازه گرفته و به عنوان خروجی ارسال می‌کند. می‌توان موارد دیگر همانند هزینه‌ی شبکه (تعداد پارامترهای به کار رفته در شبکه) و ... را نیز به تابع برازش افزود.

تابع تکامل :

```python
def EA(self, population_size, sample_size, iteration):
    for _ in range(iteration):
        sample_space = [(nasnet, self.ϕ(nasnet)) for nasnet in [NASNet(self.x0,N=self.N,f=self.f) for _ in range(population_size)]]
        sample = [sample_space[x] for x in random.sample(range(0, population_size-1), sample_size)]
        parent = sorted(sample, key=lambda tup: tup[1])[0]
        child  = self.μ(parent)
        child_fitness = self.ϕ(child)
        sample_space.pop()
        sample_space+=[(child,child_fitness)]
    return sorted(sample, key=lambda tup: tup[1])[0]
```

طبق گفته‌ی مقاله، این تابع نوعی الگوریتم تکاملی عمردار می‌باشد (اعضای پیر جامعه به تدریج می‌میرند و اعضای جدید به جامعه افزوده می‌شوند) برای ساخت اعضای جدید و افزودن آن به جامعه، با انتخاب یک نمونه‌ی با تعداد مشخص و انتخاب برازنده ترین عضو آن به عنوان والد، او را جهش داده و فرزند را تولید می‌کنیم. در ادامه با مردن مسن ترین عضو جامعه، جوان ترین عضو بدان افزوده می‌شود و این روند به تعداد مشخص که توسط ما تعیین شده ادامه می‌یابد. در پایان، برازنده ترین عضو جامعه به عنوان پاسخ برگردانده خواهد شد.

کل توابع در ۳ خط :

```
# load data
trainloader,testloader,_ = load_and_normalize_cifar10()
# init algorithm
NAS = NASAlgo(x0=trainloader[0],trainloader=trainloader, testloader=testloader,N=1,f=1)
# perform algorithm
best_net = NAS.EA(population_size=50, sample_size=10, iteration=100)
```

استفاده از الگوریتم بالا سرراست است. تنها کافیست با بارگزاری داده و پس از مقداری دهی الگوریتم، جستوجو را در فضای ساختارهای شبکه‌های CNN آغاز نماییم.

پایان

با تشکر از توجه شما.