• چکیده

اخیراً پیشرفت‌های قابل توجهی در الگوریتم‌های مبتنی بر ازدحام دیده شده است. یکی از این الگوریتم‌ها بهینه‌سازی ازدحام ذرات (PSO) است که الهام گرفته از طبیعت و مبتنی بر جمعیت است که از آن برای بهینه سازی فراپارامترهای شبکه‌های عصبی استفاده شده. در یکی از تحقیقات اخیر، الگوریتمی تحت عنوان بهینه‌سازی ازدحام ذرات چندمرحله‌ای (MPSO) ارائه شده که در این گزارش پیاده‌سازی آن صورت گرفته. از این الگوریتم برای بهینه سازی فراپارامترهای شبکه‌ی عصبی همگشتی (CNN) بهره گرفته و نتایج استفاده از آن گزارش شده است. در این الگوریتم به جای یک ازدحام، از دو مرحله ازدحام استفاده می شود ولی شباهت زیادی به همان الگوریتم PSO دارد. دقت نهایی روی دیتاست MNIST هم به عدد گزارش شده توسط خود مقاله نزدیک است.

واژه‌های کلیدی:

بهینه‌سازی، شبکه‌های عصبی، ازدحام ذرات

فهرست مطالب

صفحه

۱

فصل اول: مقدمه

برای اینکه بتوان دخالت انسانی در تنظیم فراپارامترهای[1] شبکه‌ی عصبی را به حداقل رساند روش‌های گوناگونی پیشنهاد شده است. در تحقیقات اخیر با الهام از طبیعت و با توجه به الگوریتم‌های تکاملی، روش‌هایی ارائه شده که یکی از این روش‌ها بهینه‌سازی ازدحام ذرات[2] یا به اختصار PSO بوده است.

الگوریتم PSO از رفتار پرندگان الهام گرفته شده است که به صورت گروهی به دنبال غذای خود حرکت می‌کنند. در ساختار گروهی این پرندگان، هر کدام با یک هیوریستیک نزدیکی خود به هدف را اندازه‌گیری می‌کنند. همچنین هر پرنده علاوه بر موقعیت فعلی خود، بهترین موقعیتی را که بوده ذخیره کرده و همچنین بهترین موقعیتی که از میان همه‌ی پرندگان به دست آمده نیز ذخیره می‌شود. آنگاه جهت بعدی حرکت هر پرنده با توجه به قوانین سینماتیک فیزیکی و با پارامترهایی محاسبه می شود که بردار برآیندی از دو بردار است؛ بردار اول در جهت بهترین موقعیت دیده شده بین همه‌ی پرندگان و بردار دوم در جهت بهترین موقعیت دیده شده توسط خود پرنده است.

محققان با الهام از این الگوریتم که در (Kennedy, 1995) آمده، الگوریتم بهینه‌سازی ازدحام ذرات چندمرحله‌ای[3] (MPSO) را ارائه کرده‌اند (Panigrahi, 2021). در این الگوریتم از دو ازدحام بهره گرفته می شود تا معماری CNN و فراپارامترهای آن بهبود داده شوند. لایه‌ی نهایی یک لایه‌ی کاملاً همبند[4] با فعال‌ساز softmax است که احتمال تعلق به هر کلاس را محاسبه می‌کند. مقدار فیتنس که هیوریستیک مورد نظر است همان دقت[5] شبکه‌ی عصبی است.

از جمله تحقیقات دیگری که منبع الهام این محققان بوده می‌توان به (Wong, 2012) اشاره کرده. بجز این منابع نویسندگان به مقالات دیگری نیز اشاره می‌کنند اما به طور مستقیم اعلام می‌کنند این دو منبع الهام آنها بوده است.

---

[1] hyperparameters

[2] particle swarm optimization

[3] multi-level particle swarm optimization

[4] fully connected layer

[5] accuracy

شیوه‌ی کلی کار الگوریتم به این صورت است که یک ازدحام با m ذره به صورت رندوم به طور اولیه مقداردهی می‌شود. این ازدحام، مرحله‌ی اول است که هر ذره‌ی آن شامل تعداد لایه‌های همگشتی[۶] (nC)، تعداد لایه‌های ادغام[۷] (nP) و تعداد لایه‌های کاملاً همبند (nF) می‌شود.

حال به ازای هر ذره‌ی مرحله‌ی اول n ذره در مرحله‌ی دوم با مقادیر اولیه‌ی تصادفی تولید می‌شود. در این مرحله هر ذره شامل تعداد فیلترهای لایه‌ی همگشتی (c_nf)، اندازه‌ی فیلتر (کرنل) در لایه‌ی همگشتی (c_fs)، پدینگ لایه‌ی همگشتی (c_pp)، اندازه‌ی گام[۸] لایه‌ی همگشتی (c_ss)، اندازه‌ی فیلترها در لایه‌ی حداکثر تجمع[۹] (p_fs)، اندازه‌ی گام لایه‌ی حداکثر تجمع (p_ss)، پدینگ در لایه‌ی ادغام (p_pp) و تعداد نورون‌های خروجی در لایه‌ی کاملاً همبند (op) است.

در هر ذره‌ی مرحله‌ی اول، پس از مقداردهی اولیه به سه مقدار مذکور، n ذره‌ی آن در مرحله‌ی دوم کاوش می‌شوند. هر کدام از این n ذره (که برای تفکیک آنها را ریزذره می‌نامیم) یک شبکه‌ی عصبی با مقادیر در دست ایجاد کرده و پس از آموزش روی داده‌های آموزشی و آزمایش روی داده‌های آزمایشی، دقت شبکه‌ی عصبی به عنوان مقدار فیتنس آن ریزذره در نظر گرفته می‌شود. آنگاه به اندازه‌ی t_max_2 بار این ریزذرات کاوش شده و هر بار در صورت بهتر بودن فیتنس، مقدار آن بروزرسانی می‌شود. همچنین همان طور که در شرح الگوریتم PSO مطرح شد، بهترین ریزذره که تحت عنوان global best یا به اختصار gBest هم شناخته می‌شود ذخیر شده که در نهایت بهترین ریزذره به ذره‌ی اولیه بازگردانده می‌شود. حال مقدار فیتنس آن ذره که personal best هم نامیده شده و با pBest نشان داده می‌شود، همان gBest ازدحام مرحله‌ی دوم ذره است. الگوریتم به اندازه‌ی t_max_1 بار روی این m ذره کاوش کرده و با هر بار تکرار مراحل بالا در نهایت بهترین ذره را پیدا کرده که همان ۱۱ پارامتر مورد نظر مسئله را به دست می‌دهند.

---

[۶] convolutional layers

[۷] pooling layers

[۸] stride

[۹] max pooling layer

۲ فصل دوم: شرح پیاده‌سازی

بخش اصلی پیاده سازی شامل کلاس‌های مربوط به ذرات و خود الگوریتم اصلی است که در دو فایل py. پیاده سازی شده است. محتویات این دو فایل برای تست کردن در یک نوت‌بوک در بستر گوگل‌کولب اجرا شده است. در این فصل نوت‌بوک را شرح می‌دهیم که محتویات آن در فایل‌های py. هم موجود است. آدرس دانلود این نوت‌بوک و دو فایل دیگر در پیوست آمده است.

۱-۲- کلاس Particle1

اولین ذره Particle1 است که در واقع همان ذره‌ی مرحله‌ی اول را پیاده سازی می‌کند. کانستراکتور آن به صورت زیر پیاده شده است:

```
class Particle1:
    def __init__(self, n) -> None:
        self.n = n
        self.nC = randint(1, 5)       # Number of convolutional layers (nC)
        self.nP = randint(1, self.nC) # Number of pooling layers (nP)
        self.nF = randint(1, self.nC) # Number of fully connected layers (nF)
        self.v_nC = randint(1 - self.nC, 5 - self.nC)
        self.v_nP = randint(1 - self.nP, 5 - self.nP)
        self.v_nF = randint(1 - self.nF, 5 - self.nF)
        self.nC_best = self.nC
        self.nP_best = self.nP
        self.nF_best = self.nF
        self.best_particle = None
        self.swarm_sl2 = [Particle2()] * self.n
```

مقدار n همان تعداد ذرات در مرحله‌ی اول است و nC، nP و nF همان متغیرهایی هستند که در فصل اول شرح داده شدند و مطابق (Panigrahi, 2021) نام‌گذاری شده‌اند. مقادیر اولیه‌ی آنها طبق جدول ۱ مقاله که در زیر آمده است به صورت تصادفی انتخاب می شود. همچنین همان طور که در جدول ۲ مقاله آمده است، تعداد هیچ کدام از لایه‌ها نباید از تعداد لایه‌های همگشتی فراتر برود که در کد رعایت شده است. متغیرهای با پسوند v_ سرعت‌های اولیه را مشخص می‌کنند. بازه‌ی این سرعت‌ها مطابق فرمول ۹ مقاله مشخص شده است که بیشینه و کمینه‌ها از روی همان جدول ۱ مقاله در نظر گرفته شده‌اند.

پیشوند best_ نماینده‌ی بهترین مقادیر متغیرها هستند که مجموعه‌ی آنها همان pBest را تشکیل می‌دهند. برای آنکه به فیتنس و هشت فراپارامتر دیگر دسترسی داشته باشیم، متغیر best_particle

را هم تعریف کرده‌ایم که از کلاس `Particle2` اســـت. متغیر `swarm_sl2` ذرات مرحله‌ی دوم (ریزذرات) مربوط به آن ذره را نگهداری می‌کند.

Table 1
Range of hyperparameters.

Convolution Neural Network (CNN) Architecture			
Layers	Hyperparameter	Range	
		Minimum value	Maximum value
Convolution	1. Number of convolutional layers (nC)	1	5
Pooling	2. Number of pooling layers (nP)	1	5
Fully Connected	3. Number of fully connected layers (nF)	1	5
Convolution	1. Number of filters (c_nf)	1	64
	2. Filter Size (c_fs) (odd)	1	13
	3. Padding pixels (c_pp)	0 (valid)	1 (same) $p = \frac{c\_fs-1}{2}$
	4. Stride Size (c_ss)(< c_fs)	1	5
Pooling	5. Filter Size (p_fs)(odd)	1	13
	6. Stride Size (p_ss)	1	5
	7.Padding pixels (p_pp) (< p_fs)	0 (valid)	1 (same) $p = \frac{p\_fs-1}{2}$
Fully Connected	8. Number of neurons (op)	1	1024

جدول ۱ مقاله

Table 2
Values of parameters.

Particle Swarm Optimization (PSO)	
Parameter	Value
Swarm size at Swarm Level-1 (nP≤ nC, nF ≤ nC)	5 × 3
Swarm size at Swarm Level-2	5 X nC X 8
Social coefficient (c_1)	2
Cognitive coefficient (c_2)	2
Inertia weight (ω)	as in Eq. (3)
Maximum iterations at Swarm Level-1	between 5 and 8
Maximum iterations at Swarm Level-2	5

جدول ۲ مقاله

$$V_i = \begin{cases} V_{max}, & if\ V_i > V_{max} \\ V_{min}, & if\ V_i < V_{min} \end{cases} \quad (9)$$

where $V_{max} = P_i^{max} - P_i, V_{min} = P_i^{min} - P_i$

فرمول ۹ مقاله

تابع زیر برای گرفتن فیتنس نوشته شده است. شبکه‌های عصبی در مرحله‌ی دوم آموزش داده شده و آزموده می شوند و مقدار فیتنس هم در ذرات مرحله‌ی دوم نگهداری می شود. در صورتی که ذره‌ای موجود نباشد، فیتنس ذره را $-\infty$ در نظر می‌گیریم.

```
    def getFitness(self):
        if self.best_particle:
            return self.best_particle.fitness
        return float('-inf')
```

تابع بعدی برای بروزرسانی مکان ذرات با توجه به فرمول ۱۰ مقاله نوشته شده است.

```
    def updatePosition(self):
        self.nC = int(self.nC + self.v_nC)
        self.nP = int(self.nP + self.v_nP)
        self.nF = int(self.nF + self.v_nF)
```

$$P_i = P_i + V_i \quad (10)$$

فرمول ۱۰ مقاله

تابع بعدی برای بروزرسانی سرعت ذرات نوشته شده است. این بروزرسانی با توجه به فرمول ۸ و ۹ مقاله به صورت زیر ا ست. مقادیر c1 و c2 طبق جدول ۲ مقاله برابر ۲ در نظر گرفته شده‌اند. مقادیر r1 و r2 در خود مقاله ذکر نشده‌اند، ولی در الگوریتم PSO مقدار تصادفی بین ۰ و ۱ دارند که در اینجا هم همین گونه در نظر گرفته شده‌اند. مقادیر بیشینه برای nF و nP هم مطابق جدول ۲ مقاله برابر nC هستند.

```
    def updateVelocity(self, w, c1=2, c2=2):
        r1 = random()
        r2 = random()
        # updating v_nC
        self.v_nC = w * self.v_nC + c1 * r1 * (self.nC_best - self.nC) + c2 *
r2 * (self.nC_best - self.nC)
        v_nC_max = 5 - self.nC
        if self.v_nC > v_nC_max:
            self.v_nC = v_nC_max
        v_nC_min = 1 - self.nC
```

```
        if self.v_nC < v_nC_min:
            self.v_nC = v_nC_min
        # updating v_nP
        self.v_nP = w * self.v_nP + c1 * r1 * (self.nP_best - self.nP) + c2 *
r2 * (self.nP_best - self.nP)
        v_nP_max = self.nC - self.nP
        if self.v_nP > v_nP_max:
            self.v_nP = v_nP_max
        v_nP_min = 1 - self.nP
        if self.v_nP < v_nP_min:
            self.v_nP = v_nP_min
        # updating v_nF
        self.v_nF = w * self.v_nF + c1 * r1 * (self.nF_best - self.nF) + c2 *
r2 * (self.nF_best - self.nF)
        v_nF_max = self.nC - self.nF
        if self.v_nF > v_nF_max:
            self.v_nF = v_nF_max
        v_nF_min = 1 - self.nF
        if self.v_nF < v_nF_min:
            self.v_nF = v_nF_min
```

$$V_i = \omega V_i + c_1 r_1 (pbest_i - P_i) + c_2 r_2 (gbest - P_i) \qquad (8)$$

فرمول ۸ مقاله

مقدار w همان ω است که از تابع زیر محاسبه می‌شود و بر پایه‌ی فرمول ۳ نوشته شده است.

```
def calculate_omega(t, t_max, a=0.2):
    if t < a * t_max:
        return 0.9
    return 1 / (1 + e ** ((10 * t - t_max) / t_max))
```

$$\omega(t) = \begin{cases} 0.9 \ when \ t < \alpha t_{max} \\ \frac{1}{1+ e^{(10t-2t_{max})/t_{max}}} \ otherwise \end{cases} \qquad (3)$$

فرمول ۳ مقاله

مقدار α با توجه به متن مقاله که در زیر آمده برابر ۰٫۲ در نظر گرفته شده است.

In its formulation symbol "t" denotes current iteration, t_{max} is the maximum number of iterations, the value of α is set to 0.2 which has shown significant improvement in results as mentioned in paper [18].

تابع بعدی مطابق آنچه که در مقدمه گفته شد pBest را برای خود ذره محاسبه می‌کند. مقدار بیشینه‌ی پیمایش روی ذرات مرحله‌ی دوم یعنی t_max کمتر و برابر ۳ در نظر گرفته شده تا زمان اجرا کمتر شود. اما می‌توان آن را بین ۵ تا ۸ مطابق جدول ۲ در نظر گرفت. طرز کار تابع به این صورت است که به ازای هر ذره‌ی مرحله‌ی دوم، یک شبکه‌ی عصبی پیاده سازی می‌کند. در ابتدا لایه‌های همگشتی و ادغام یکی در میان قرار گرفته و قبل از قرار گرفتن لایه‌ی کاملاً همبند (Dense)، لایه‌ی Flatten هم قرار داده می‌شود. همچنین نخستین لایه یک ورودی به صورت input_shape دریافت می‌کند که چون ما از دیتاست MNIST برای آزمایش استفاده کرده‌ایم و داده‌های آن تک‌رنگ هستند، کانال ورودی ۱ و اندازه‌ی تصاویر ۲۸ در ۲۸ بوده که با سه تایی (1, 28, 28) تبیین می‌شود. همچنین از لایه‌های Dropout با نرخ ۰٫۲ و فعال‌سازهای ReLU بجز لایه‌ی خروجی که softmax است و batch_size=128 طبق متن زیر از مقاله استفاده شده است.

of MPSO–CNN. All runs take 128 mini-batch size, RELU activation function and dropout rate of 20%. CNN generated features are used in softmax layer for classification of images.

```
    def calculate_pbest(self, number_of_classes, x_train, y_train, x_test,
y_test): # based on algorithm 3
        cnn_counter = 0
        cnns_trained_time = 0
        best_particle = None
        # t_max = 5
        t_max = 3
        for t in range(t_max):
            for particle in self.swarm_sl2:
                c_nf = particle.c_nf
                c_fs = particle.c_fs if particle.c_fs % 2 == 1 else
particle.c_fs - 1
                c_pp = particle.c_pp
                c_ss = particle.c_ss
                p_fs = particle.p_fs
                p_ss = particle.p_ss
                p_pp = particle.p_pp
                op   = particle.op
                cnn = Sequential()
                nC_counter = 0
                nP_counter = 0
                try:
                    print('cnn')
```

```
                while nP_counter != self.nP and nC_counter != self.nC:
                    if nC_counter != self.nC:
                        if nC_counter == 0:
                            cnn.add(Conv2D(c_nf,
                                           c_fs,
                                           padding=m_padding[c_pp],
                                           strides=c_ss,
                                           activation='relu',
                                           input_shape=(28, 28, 1)))
                        else:
                            cnn.add(Conv2D(c_nf * 2 ** (nC_counter),
                                           c_fs,
                                           padding=m_padding[c_pp],
                                           strides=c_ss,
                                           activation='relu'))
                        nC_counter += 1
                    if nP_counter != self.nP:
                        cnn.add(MaxPooling2D(pool_size=p_fs,
                                             strides=p_ss,
                                             padding=m_padding[p_pp]))
                        nP_counter += 1
                cnn.add(Flatten())
                for _ in range(self.nF):
                    cnn.add(Dropout(0.2))
                    cnn.add(Dense(op, activation='relu'))
                cnn.add(Dropout(0.2))
                cnn.add(Dense(number_of_classes, activation='softmax'))
                start_time = time.time()
                cnn.compile(loss='categorical_crossentropy',
optimizer='adam', metrics='accuracy')
                cnn.fit(x_train, y_train, batch_size=128)
                cnns_trained_time += time.time() - start_time
                cnn_counter += 1
                loss, accuracy = cnn.evaluate(x_test, y_test)
            except:
                accuracy = float('-inf')
            if accuracy > particle.fitness:
                particle.fitness = accuracy
                particle.c_nf_best = c_nf
                particle.c_fs_best = c_fs
                particle.c_pp_best = c_pp
                particle.c_ss_best = c_ss
                particle.p_fs_best = p_fs
                particle.p_ss_best = p_ss
                particle.p_pp_best = p_pp
```

```
                particle.op_best = op
            if not best_particle or accuracy > best_particle.fitness:
                best_particle = particle
            w = calculate_omega(t, t_max)
            particle.updateVelocity(w, best_particle)
            particle.updatePosition()
    return best_particle, cnn_counter, cnns_trained_time
```

نکته و چالش مهمی که در پیاده‌سازی وجود داشت این بود که گاهی اوقات چینش شبکه‌ی عصبی درست نبود و فراپارامترها و لایه‌های ادغام همخوانی نداشتند. در نتیجه لایه‌های ادغام متداول کاهش ابعاد زیادی را در پی داشته و شبکه‌ی عصبی با مشکل مواجه می‌شد. در این حالت فیتنس را برابر ∞- قرار می‌دهیم. در ادامه در صورتی که فیتنس بهتر از بهترین فیتنس خود ذره‌ی مرحله‌ی دوم شده باشد، بروزرسانی می‌شود. همچنین اگر بهتر از gBest کل ازدحام دوم شده باشد، gBest هم بروزرسانی می‌شود و به همراه آنها مقادیر متناظر. مقدار gBest در `best_partcile` ذخیره شده و به عنوان pBest به ذره‌ی اول برگردانده می‌شود. منطق این تابع مطابق الگوریتم ۳ مقاله که در زیر آمده است می‌باشد.

Algorithm 3
Hybrid MPSO-CNN algorithm at swarm level-2.

Input: particle (P_i) of swarm level-1, maximum number of iterations ($tmax^2$) and search space for hyperparameters
Output: (CNN hyperparameters, fitness value))

Algorithm:
for each particle j = 1 to n of swarm at level-2 do
    initialize particle's position in specified range: [c_nf, c_fs, c_pp, c_ss, p_fs, p_ss, p_fs, op]
    setup a CNN model: CNN(P_i, P_{ij})
    compute fitness value using CNN: F_{ij}
    initialize personal best: $pbest_{ij}$
    initialize global best: $gbest_i$
end
while (maximum number of iterations is not reached: $tmax^2$)
    for each particle j = 1 to n of swarm at level-2 do
        update particle's velocity and position: (V_{ij}, P_{ij})
        setup a CNN model: CNN(P_i, P_{ij})
        compute fitness value using CNN: F'_{ij}
        update personal best: $pbest_{ij}$
        update global best: $gbest_i$
    end
end
return((P_i, $gbest_i$), CNN(P_i, $gbest_i$))

الگوریتم ۳ مقاله

۲-۲- کلاس `Particle2`

این کلاس همان ذرات مرحله‌ی دوم را پیاده‌سازی می‌کند که کانستراکتور آن به صورت زیر است:

```
class Particle2:
    def __init__(self):
        self.c_nf   = randint(1,
64)                                          # Number of filters (c_nf)
```

```
        self.c_fs   = randint(1,
13)                                             # Filter Size (c_fs) (odd)
        self.c_pp   = randint(0,
1)                                              # Padding pixels (c_pp)
        self.c_ss   = randint(1, 5) if self.c_fs > 5 else randint(1,
self.c_fs) # Stride Size (c_ss)(<c_fs)
        self.p_fs   = randint(1,
13)                                             # Filter Size (p_fs)(odd)
        self.p_ss   = randint(1,
5)                                              # Stride Size (p_ss)
        self.p_pp   = randint(0,
1)                                              # Padding pixels (p_pp)
        self.op     = randint(1,
1024)                                           # Number of neurons (op)
        self.v_c_nf = randint(1 - self.c_nf, 64 - self.c_nf)
        self.v_c_fs = randint(1 - self.c_fs, 13 - self.c_fs)
        self.v_c_pp = randint(0 - self.c_pp, 1 - self.c_pp)
        if self.c_fs > 5:
            self.v_c_ss = randint(1 - self.c_ss, 4 - self.c_ss)
        else:
            self.v_c_ss = randint(1 - self.c_ss, self.c_fs - self.c_ss)
        self.v_p_fs = randint(1 - self.p_fs, 13 - self.p_fs)
        self.v_p_ss = randint(1 - self.p_ss, 5 - self.p_ss)
        if self.p_fs > 2:
            self.v_p_pp = randint(0 - self.p_pp, 1 - self.p_pp)
        else:
            self.v_p_pp = randint(0 - self.p_pp, self.p_fs - self.p_pp)
        self.v_op = randint(1 - self.op, 1024 - self.op)
        self.c_nf_best = self.c_nf
        self.c_fs_best = self.c_fs
        self.c_pp_best = self.c_pp
        self.c_ss_best = self.c_ss
        self.p_fs_best = self.p_fs
        self.p_ss_best = self.p_ss
        self.p_pp_best = self.p_pp
        self.op_best = self.op
        self.fitness = float('-inf')
```

منطق کانســـتراکتور مطابق کلاس اول بوده و با اســـتفاده از همان جداول و مطابق خود مقاله نامگذاری متغیرها صورت گرفته است. همچنین متغیر `fitness` در ابتدا برابر $-\infty$ در نظر گرفته می‌شود.

دو تابع دیگر این کلاس هم منطقی مشـــابه با کلاس ذره‌ی اول دارند. باید توجه داشـــت که فرمول‌های استفاده شده هم مشابه هستند؛ یعنی به عنوان مثال به جای فرمول ۸، ۹ و ۱۰ مقاله متناظراً از فرمول‌های ۵، ۶ و ۷ استفاده شده است.

$$V_{ij} = \omega V_{ij} + c_1 r_1 (pbest_{ij} - P_{ij}) + c_2 r_2 (gbest_i - P_{ij} \quad (5)$$

$$V_{ij} = \begin{cases} V_{max}, & if\ V_{ij} > V_{max} \\ V_{min}, & if\ V_{ij} < V_{min} \end{cases} \quad (6)$$

$$\text{where } V_{max} = P_{ij}^{max} - P_{ij}, V_{min} = P_{ij}^{min} - P_{ij}$$

$$P_{ij} = P_{ij} + V_{ij} \quad (7)$$

فرمول‌های ۵، ۶ و ۷ مقاله

```
    def updatePosition(self):
        self.c_nf = int(self.c_nf + self.v_c_nf)
        self.c_fs = int(self.c_fs + self.v_c_fs)
        self.c_pp = int(self.c_pp + self.v_c_pp)
        self.c_ss = int(self.c_ss + self.v_c_ss)
        self.p_fs = int(self.p_fs + self.v_p_fs)
        self.p_ss = int(self.p_ss + self.v_p_ss)
        self.p_pp = int(self.p_pp + self.v_p_pp)
        self.op   = int(self.op + self.v_op)

    def updateVelocity(self, w, best_particle, c1=2, c2=2):
        r1 = random()
        r2 = random()
        # updating v_c_nf
        self.v_c_nf = w * self.v_c_nf + c1 * r1 * (self.c_nf_best -
self.c_nf) + c2 * r2 * (best_particle.c_nf_best - self.c_nf)
        v_c_nf_max = 64 - self.c_nf
        if self.v_c_nf > v_c_nf_max:
            self.v_c_nf = v_c_nf_max
        v_c_nf_min = 1 - self.c_nf
        if self.v_c_nf < v_c_nf_min:
            self.v_c_nf = v_c_nf_min
        # updating v_c_fs
        self.v_c_fs = w * self.v_c_fs + c1 * r1 * (self.c_fs_best -
self.c_fs) + c2 * r2 * (best_particle.c_fs_best - self.c_fs)
        v_c_fs_max = 13 - self.c_fs
        if self.v_c_fs > v_c_fs_max:
            self.v_c_fs = v_c_fs_max
```

```
        v_c_fs_min = 1 - self.c_fs
        if self.v_c_fs < v_c_fs_min:
            self.v_c_fs = v_c_fs_min
        # updating v_c_pp
        self.v_c_pp = w * self.v_c_pp + c1 * r1 * (self.c_pp_best -
self.c_pp) + c2 * r2 * (best_particle.c_pp_best - self.c_pp)
        v_c_pp_max = 1 - self.c_pp
        if self.v_c_pp > v_c_pp_max:
            self.v_c_pp = v_c_pp_max
        v_c_pp_min = 0 - self.c_pp
        if self.v_c_pp < v_c_pp_min:
            self.v_c_pp = v_c_pp_min
        # updating v_c_ss
        self.v_c_ss = w * self.v_c_ss + c1 * r1 * (self.c_ss_best -
self.c_ss) + c2 * r2 * (best_particle.c_ss_best - self.c_ss)
        v_c_ss_max = self.c_fs - 1 - self.c_ss
        if self.v_c_ss > v_c_ss_max:
            self.v_c_ss = v_c_ss_max
        v_c_ss_min = 1 - self.c_ss
        if self.v_c_ss < v_c_ss_min:
            self.v_c_ss = v_c_ss_min
        # updating v_p_fs
        self.v_p_fs = w * self.v_p_fs + c1 * r1 * (self.p_fs_best -
self.p_fs) + c2 * r2 * (best_particle.p_fs_best - self.p_fs)
        v_p_fs_max = 13 - self.p_fs
        if self.v_p_fs > v_p_fs_max:
            self.v_p_fs = v_p_fs_max
        v_p_fs_min = 1 - self.p_fs
        if self.v_p_fs < v_p_fs_min:
            self.v_p_fs = v_p_fs_min
        # updating v_p_ss
        self.v_p_ss = w * self.v_p_ss + c1 * r1 * (self.p_ss_best -
self.p_ss) + c2 * r2 * (best_particle.p_ss_best - self.p_ss)
        v_p_ss_max = self.p_fs - 1 - self.p_ss # right ?
        v_p_ss_max = 5 - self.p_ss
        if self.v_p_ss > v_p_ss_max:
            self.v_p_ss = v_p_ss_max
        v_p_ss_min = 1 - self.p_ss
        if self.v_p_ss < v_p_ss_min:
            self.v_p_ss = v_p_ss_min
        # updating v_p_pp
        self.v_p_pp = w * self.v_p_pp + c1 * r1 * (self.p_pp_best -
self.p_pp) + c2 * r2 * (best_particle.p_pp_best - self.p_pp)
        v_p_pp_max = 1 - self.p_pp
        if self.v_p_pp > v_p_pp_max:
```

```
            self.v_p_pp = v_p_pp_max
        v_p_pp_min = 0 - self.p_pp
        if self.v_p_pp < v_p_pp_min:
            self.v_p_pp = v_p_pp_min
        # updating v_op
        self.v_op = w * self.v_op + c1 * r1 * (self.op_best - self.op) + c2 *
r2 * (best_particle.op_best - self.op)
        v_op_max = 1024 - self.op
        if self.v_op > v_op_max:
            self.v_op = v_op_max
        v_op_min = 1 - self.op
        if self.v_op < v_op_min:
            self.v_op = v_op_min
```

۳-۲- کلاس HybridMPSOCNN

این تابع عملکرد اصلی الگوریتم را پیاده‌سازی می‌کند. کانستراکتور آن به صورت زیر است.

```
class HybridMPSOCNN:
    def __init__(self, number_of_classes, x_train, y_train, x_test, y_test,
m=5, n=8):
        self.m = m
        self.n = n
        self.number_of_classes = number_of_classes
        self.x_train = x_train
        self.y_train = y_train
        self.x_test = x_test
        self.y_test = y_test
        self.number_of_trained_cnns = 0
        self.cnns_trained_time = 0
        self.gbest = None
```

number_of_classes تعداد برچسب‌های داده‌های هدف را مشخص می‌کند. مقادیر m و n به ترتیب تعداد ذرات مرحله‌ی اول و دوم هستند. متغیر gbest در خود gBest کل ذرات مرحله‌ی اول را ذخیره می‌کند که پاسخ نهایی هست. دو متغیر number_of_trained_cnns و cnns_trained_cnns برای آمار نهایی ایجاد شده‌اند.

تابع بعدی آمار نهایی مربوط به بهترین متغیرها و بهترین دقت به دست آمده و همچنین تعداد شبکه‌های آموزش داده شده و میانگین زمان آموزش و آزمایش شبکه‌ها را می‌دهد.

```
    def summary(self):
        print(f'Number of trained cnn\'s: {self.number_of_trained_cnns}')
```

```
        print(f'Average cnn train time: {self.cnns_trained_time /
self.number_of_trained_cnns}')
        print('Best Hyperparameters:')
        print(f'Number of convolutional layers (nC): {self.gbest.nC_best}')
        print(f'Number of pooling layers (nP): {self.gbest.nP_best}')
        print(f'Number of fully connected layers (nF): {self.gbest.nF_best}')
        print(f'Number of filters (c_nf):
{self.gbest.best_particle.c_nf_best}')
        print(f'Filter Size (c_fs) (odd):
{self.gbest.best_particle.c_fs_best}')
        print(f'Padding pixels (c_pp): {self.gbest.best_particle.c_pp_best}')
        print(f'Stride Size (c_ss)(< c_fs):
{self.gbest.best_particle.c_ss_best}')
        print(f'Filter Size (p_fs)(odd):
{self.gbest.best_particle.p_fs_best}')
        print(f'Stride Size (p_ss): {self.gbest.best_particle.p_ss_best}')
        print(f'Padding pixels (p_pp) (< p_fs):
{self.gbest.best_particle.p_pp_best}')
        print(f'Number of neurons (op): {self.gbest.best_particle.op_best}')
        print(f'Fitness (aka accuracy of best model):
{self.gbest.getFitness()}')
```

تابع بعدی که در واقع هسته‌ی الگوریتم ا ست در ابتدا m ذره‌ی مرحله‌ی اول را ایجاد کرده و به اندازه‌ی t_max که در اینجا هم باز ۳ در نظر گرفته شده روی این ذرات پیمایش را انجام می‌دهد.

```
    def run(self): # based on algorithm 2
        swarm1 = [Particle1(self.n)] * self.m
        # t_max = randint(5, 8)
        t_max = 3
        for t in range(t_max):
            for particle in swarm1:
                nC = particle.nC
                nP = particle.nP
                nF = particle.nF
                best_particle, cnn_counter, cnns_trained_time =
particle.calculate_pbest(self.number_of_classes, self.x_train, self.y_train,
self.x_test, self.y_test)
                self.number_of_trained_cnns += cnn_counter
                self.cnns_trained_time += cnns_trained_time
                print(f'number of trained cnn\'s so far:
{self.number_of_trained_cnns}')
                if best_particle.fitness > particle.getFitness():
                    particle.nC_best = nC
                    particle.nP_best = nP
                    particle.nF_best = nF
```

```
                    particle.best_particle = best_particle
                if not self.gbest or best_particle.fitness >
self.gbest.getFitness():
                    self.gbest = particle
                w = calculate_omega(t, t_max)
                particle.updateVelocity(w)
                particle.updatePosition()
        print('Done!')
```

در هر پیمایش pBest هر ذره انتخاب شـــده و باز هم مثل پیمایش دیگری که داشـــتیم، pBest هر ذره و gBest کل ذرات در صورت بهتر بودن بروزرسانی می‌شوند. منطق کد بالا بر اساس الگوریتم ۲ مقاله است که در زیر آمده.

Algorithm 2
Hybrid MPSO-CNN algorithm at swarm level-1.

Input: maximum number of iterations ($tmax^1$) and search space for hyperparameters
Output: (CNN hyperparameters: *[nC, nP, nF, c_nf, c_fs, c_pp, c_ss, p_fs, p_ss, p_fs, op]*, fitness value)

Algorithm:
initialize particle's position vector in specified range: [nC, nP, nF]
while (maximum number of iterations is not reached: $tmax^1$)
Calculate ω using Eq. (3)
    for each particle i = 1 to m of swarm at level-1 do
        find hyperparameters and its fitness value (as in Algorithm 3)
        update fitness value: F_i =fitness<P_i > = fitness (P_i, $gbest_i$)
        update personal best: $pbest_i$
        update global best: gbest
        update particle's velocity and position: (V_i, P_i)
    end
end
return(gbest, CNN (gbest))

الگوریتم ۲ مقاله

۳

فصل سوم: جمع‌بندی و نتیجه‌گیری

برای آزمایش دو تســـت انجام شـــد که در هر دوی تســـت‌ها مقادیر m=3 و n=3 برای کاهش زمان اجرا انتخاب شـدند. خلاصـه گزارش این دو بار اجرا که حدود سـه سـاعت هر کدام زمان برد در نوت‌بوک که از پیوست قابل دریافت است و در زیر آمده. دیتاست استفاده شده MNIST بوده است که پیش‌پردازش هم شده.

اطلاعات این دیتاســت در هنگام پیش‌پردازش در نوت‌بوک گرفته شــده که متشــکل از ۶۰۰۰۰ داده‌ی آموزشی و ۱۰۰۰۰ داده‌ی آزمایشی است. داده‌های دیتاست به صورت تصاویر grayscale هستند که به همین دلیل فقط یک کانال دارند.

```
optimizer.summary()
```

Python

```
Number of trained cnn's: 81
Average cnn train time: 185.6814292948923
Best Hyperparameters:
Number of convolutional layers (nC): 3
Number of pooling layers (nP): 1
Number of fully connected layers (nF): 1
Number of filters (c_nf): 37
Filter Size (c_fs) (odd): 5
Padding pixels (c_pp): 1
Stride Size (c_ss)(< c_fs): 1
Filter Size (p_fs)(odd): 7
Stride Size (p_ss): 2
Padding pixels (p_pp) (< p_fs): 1
Number of neurons (op): 463
Fitness (aka accuracy of best model): 0.9865000247955322
```

بهترین دقت گزارش شــده در تســت اول 98.6% بوده که شــرح مربوط به فراپارامترهای آن نیز در بالا آمده است. در این تست از هیچ بهینه‌سازی استفاده نشده است.

```
optimizer.summary()
```

Python

```
Number of trained cnn's: 81
Average cnn train time: 156.0086041909677
Best Hyperparameters:
Number of convolutional layers (nC): 5
Number of pooling layers (nP): 1
Number of fully connected layers (nF): 1
Number of filters (c_nf): 62
Filter Size (c_fs) (odd): 7
Padding pixels (c_pp): 1
Stride Size (c_ss)(< c_fs): 1
Filter Size (p_fs)(odd): 8
Stride Size (p_ss): 3
Padding pixels (p_pp) (< p_fs): 1
Number of neurons (op): 10
Fitness (aka accuracy of best model): 0.9672999978865491
```

همان طور که مشاهده می شود در تست دوم بهترین دقت 96.7% بوده که نسبت به تست اول کمتر است. همان طور که در مقاله هم ذکر شده نمی‌توان همیشه انتظار دقت بالا و یا رسیدن به دقت معقول را داشت. اما باز هم این دقت به دست آمده قابل قیاس با مقایسه‌هایی است که در خود مقاله انجام گرفته است. در تست دوم از بهینه‌ساز آدام[1] استفاده شد.

یکی از نکات قابل توجه تعداد شبکه‌های عصبی است که در هر دو ۸۱ است، که همان چیزی است که انتظار داریم، یعنی t_max_1 * m * t_max_2 * n = 3 * 3 * 3 * 3 = 81. یعنی در این دو تست هیچ کدام از شبکه‌های عصبی با مشکل مواجه نشدند. اما شاید در تست‌های دیگر تعداد شبکه‌های عصبی کمتر باشد.

[1] adam optimizer

۴ منابع و مراجع

Kennedy, J. a. (1995). Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks* (Vol. 4, pp. 1942-1948 vol.4).

Panigrahi, P. S. (2021). Hybrid MPSO-CNN: Multi-level Particle Swarm optimized hyperparameters of Convolutional Neural Network. *Swarm and Evolutionary Computation, 63*, 100863. doi:https://doi.org/10.1016/j.swevo.2021.100863

Wong, S. L. (2012). A hybrid particle swarm optimization and its application in neural networks. *Expert Systems with Applications, 39*, 395-405. doi:https://doi.org/10.1016/j.eswa.2011.07.028