



گزارش کار پروژه کارشناسی

بهینه سازی ابر پارامتر ها در یادگیری عمیق با استفاده از الگوریتم ژنتیک

دانشجو: پوریا شجاعی



2	خلاصه:
3	بارگذاری داده‌ها:
3	مرحله پیش پردازش (preprocessing):
4	ساخت مدل شبکه عصبی:
5	مقدار دهی اولیه (initialization):
6	تابع ساخت جمعیت اولیه (generation_population):
6	ارزیابی تناسب مدل (fitness_evaluation):
6	تابع انتخاب (selection):
7	تابع ترکیب (crossover):
8	تابع جهش (mutation):
9	تابع اصلی (main):

خلاصه :

در این پروژه ما به بررسی نقش الگوریتم ژنتیک در بهینه سازی ابر پارامتر ها در یادگیری عمیق پرداختیم و سعی کردیم با استفاده از الگوریتم ژنتیک سرعت انجام الگوریتم ها فضای جستجو را بهبود ببخشیم .

در برنامه با استفاده از فریم ورک keras و مجموعه داده های عددی mnist با استفاده از شبکه های عصبی پیاده سازی میکنیم که در ادامه به توضیح هر قسمت میپردازیم .

بارگذاری داده‌ها:

در این قسمت با استفاده از دستور زیر داده‌ها را به چهار بخش تقسیم می‌کنیم.

همانگونه که مشخص است از قسمت `train` برای آموزش داده‌ها و از قسمت `test` برای ارزیابی نهایی استفاده می‌کنیم.

```
#loading datasets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

مرحله پیش پردازش (preprocessing):

در این قسمت باید داده‌های مورد نظر را تغییر اندازه و تغییر مقایس دهیم و این کار را با استفاده از دستورات زیر انجام می‌دهیم.

```
# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
```

```
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
```

در مرحله بعد با دستورات زیر مقادیر خروجی را دسته بندی می‌کنیم.

```

num_classes=10
input_shape = (28, 28, 1)
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

```

ساخت مدل شبکه عصبی:

در این قسمت با استفاده از فریم ورک keras مدل اولیه را میسازیم و مقدار ابر پارامترها را به آن پاس میدهیم و مقدار گذاری آن را به الگوریتم ژنتیک می دهیم. وان را کامپایل و بر روی دادهای تست ارزیابی میکنیم و مدل اولیه را برمیگردانیم.

```

def CNN_MODEL(f1,f2,a1,a2,k,d,opt,ep):
    model = keras.Sequential(
        [
            keras.Input(shape=(28,28,1)),
            layers.Conv2D(f1, kernel_size=(k, k), activation=a1),
            layers.MaxPooling2D(pool_size=(2, 2)),
            layers.Conv2D(f2, kernel_size=(k, k), activation=a2),
            layers.MaxPooling2D(pool_size=(2, 2)),
            layers.Flatten(),
            layers.Dropout(d),
            layers.Dense(10, activation="softmax"),
        ]
    )
    model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])

    model.fit(x_train, y_train, batch_size=100, epochs=ep, validation_split=0.1)
    return model

```

مقدار دهی اولیه (initialization):

برای مقدار دهی اولیه از تابع زیر استفاده میکنیم و هر متغیر را از بین چند مقدار بصورت تصادفی انتخاب میکند و یک دیکشنری را به که حاوی مقادیر تابع است را انتقال میدهیم.

```
def initialization():
    parameters = {}
    #filter
    f1 = choice([32, 64])
    parameters["f1"] = f1
    f2 = choice([64, 128])
    parameters["f2"] = f2

    #kernel size
    k = choice([3,5])
    parameters["k"] = k

    #activation function
    a1 = choice(["relu", "tanh","selu", "elu"])
    parameters["a1"] = a1
    a2 = choice(["relu","tanh" ,"selu", "elu"])
    parameters["a2"] = a2

    #dropout
    d = round(uniform(0.1, 0.5), 1)
    parameters["d"] = d

    #optimizer
    opt = choice(["adamax", "adadelata", "adam", "adagrad"])
    parameters["opt"] = opt

    #number of epoches
    ep = randint(7,10)
    parameters["ep"] = ep

    return parameters
```

تابع ساخت جمعیت اولیه (generation_population):

این تابع یک عدد را از کاربر میگیرد که نشان دهنده جمع اولیه مورد نیاز است و سپس به تعداد آن تابع مقدار دهی اولیه را فراخوانی میکند و مقدارها را میگرد و درون متغیر کروموزوم میگذارد به در لیست جمعیت اضافه میکند و لیست را به تابع اصلی بر میگرداند.

```
def generate_population(n):
    population = []
    for i in range(n):
        chromosome = initialization()
        population.append(chromosome)
    return population
```

ارزیابی تناسب مدل (fitness_evaluation):

برای ارزیابی تناسب مدل از تابع از دستور زیر استفاده میکنیم و مدل را که از مراحل قبل آماده کردیم و مقادیر اولیه را به آن دادیم و آن را کامپایل و فیت کردیم را بر روی قسمت x_test, y_test انجام میدهیم.

```
# Fitness evaluation metric: Classification Accuracy
def fitness_evaluation(model):
    metrics = model.evaluate(x_test, y_test)
    return metrics[1]
```

تابع انتخاب (selection):

در این تابع ما اول لیست تمام مقادیر تابع تناسب را دریافت میکنیم و با استفاده از متد چرخ رولت دو مقدار را انتخاب میکنیم و شماره آن دو والد را به تابع اصلی بر میگردانیم

شیوه ی کار ان به این صورت است که اول با توجه به مقدار تناسب ان ها درصد شانس به ان ها میدهد بعد با مقدار درصد هر کدام به ان مقدار شانس میدهد و بعد با استفاده از تابع تصادفی یک شانس از هر تابع انتخاب میشود و شماره ان دو والد به تابع برگردانده میشوند.

```
def selection(pop_fitness):  
    total = sum(pop_fitness)  
    percentage = [round((x/total) * 100) for x in pop_fitness]  
    selection_wheel = []  
    for pop_index,num in enumerate(percentage):  
        selection_wheel.extend([pop_index]*num)  
    parent1_ind = choice(selection_wheel)  
    parent2_ind = choice(selection_wheel)  
    return [parent1_ind, parent2_ind]
```

تابع ترکیب (crossover):

بعد ایجاد جمعیت اولیه و ارزیابی انها دوتا از والدین که تناسب بهتری دارند انتخاب میشوند و به تابع ترکیب فرستاده میشوند و مقادیر انها بصورت رندم جابه جابه میشوند و دوباره به تابع اصلی فرستاده میشوند.


```

def crossover(parent1, parent2):
    child1 = {}
    child2 = {}

    child1["f1"] = choice([parent1["f1"], parent2["f1"]])
    child1["f2"] = choice([parent1["f2"], parent2["f2"]])

    child2["f1"] = choice([parent1["f1"], parent2["f1"]])
    child2["f2"] = choice([parent1["f2"], parent2["f2"]])

    child1["a1"] = parent1["a2"]
    child2["a1"] = parent2["a2"]

    child1["a2"] = parent2["a1"]
    child2["a2"] = parent1["a1"]

    child1["k"] = choice([parent1["k"], parent2["k"]])
    child2["k"] = choice([parent1["k"], parent2["k"]])

    child1["opt"] = parent2["opt"]
    child2["opt"] = parent1["opt"]

    child1["ep"] = parent1["ep"]
    child2["ep"] = parent2["ep"]
    return [child1, child2]

```

تابع جهش (mutation):

در این تابع ما والدین را از تابع اصلی میگیریم و روی یک قسمت آن بصورت رندم تغییراتی انجام میدهیم. در

این مثال ما تغییرات را بر روی نرخ تکرار انجام میدهیم اما بسته به نیاز میتوان هر کدام از قسمت ها را دچار جهش کرد.

```

def mutation(chromosome):
    flag = randint(0,40)
    if flag <= 20:
        chromosome["ep"] += randint(0, 10)
    return chromosome

```

تابع اصلی (main) :

در این قسمت ما مقادیر اولیه مانند generations که تعداد نسل ها را نشان می دهد و threshold که حد مجاز برای توقف تابع با توجه به تناسب در هر قسمت و num_pop که تعداد جمعیت اولیه را نشان میدهد سپس جمعیت اولیه ساخته میشود و در population قرار داده میشود در مرحله بعد با توجه به تعداد نسل ها حلقه تکرار میشود و در هر تکراره اطلاعات را از هر کروموزوم بازیابی میکند و در متغیر های مربوطه قراره میدهد وان را به مدل پاس میدهد و مدل ارزیابی میشود و مقادیر را بازمیگرداند و تناسب ها در لیست pop_fitness ذخیر میکند در مرحله بعد با توجه به مقادیر تناسب و تابع selection دو والد انتخاب و به مرحله ترکیب میروند و بعد به مرحله جهش و فرزندان به لیست population اضافه میشوند و در اخر هر مرحله بهترین جواب ها نمایش داده میشوند و در مرحله اخر هم بدترین والد ها از جمعیت اولیه حذف میشود.

```

generations = 3
threshold = 90
num_pop = 4

population = generate_population(num_pop)

for g in range(generations):
    pop_fitness = []
    for chromosome in population:
        f1 = chromosome["f1"]
        f2 = chromosome["f2"]
        a1 = chromosome["a1"]
        a2 = chromosome["a2"]
        k = chromosome["k"]
        d = chromosome["d"]
        opt = chromosome["opt"]
        ep = chromosome["ep"]

        try:
            model = CNN_MODEL(f1, f2, a1, a2, k, d, opt, ep)
            acc = fitness_evaluation(model)
            print("Parameters: ", chromosome)
            print("Accuracy: ", round(acc, 3))
        except:
            acc=0
            print("Parameters: ", chromosome)
            print("Invalid parameters - Build fail")

    pop_fitness.append(acc)

```

```

print(pop_fitness)

parents_ind = selection(pop_fitness)
parent1 = population[parents_ind[0]]
parent2 = population[parents_ind[1]]

children = crossover(parent1, parent2)
child1 = mutation(children[0])
child2 = mutation(children[1])

population.append(child1)
population.append(child2)

print("Generation ", g+1, " Outcome: ")
if max(pop_fitness) >= threshold:
    print("Obtained desired accuracy: ", max(pop_fitness))
    sys.exit()
else:
    print("Maximum accuracy in generation {} : {}".format(g+1, max(pop_fitness)))

first_min = min(pop_fitness)
first_min_ind = pop_fitness.index(first_min)
population.remove(population[first_min_ind])
second_min = min(pop_fitness)
second_min_ind = pop_fitness.index(second_min)
population.remove(population[second_min_ind])

```

و این تابع را هر بار که اجرا کنیم نتایج مختلفی میدهد با توجه به رندم بودن مقادیر اولیه و تغییرات ترکیب و جهش برای اطمینان بیشتر از جواب میتوانیم تعداد نسل ها و تعداد تکرار ها را افزایش دهیم تا از جواب مطمئن شویم .

