UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Progetto Laboratorio di Intelligenza Artificiale

Francesco Maria
Russo
Matr. 526864

ANNO ACCADEMICO 2023/2024

Indice

1 Introduzione

Nel contesto del riconoscimento di immagini, l'ottimizzazione delle architetture di reti neurali convoluzionali (CNN) è essenziale per garantire elevate prestazioni. Questo progetto si propone di implementare un algoritmo genetico in Python utilizzando TensorFlow per ottimizzare l'architettura di una rete neurale dedicata al riconoscimento di immagini. L'obiettivo principale è massimizzare l'efficienza della rete considerando quattro aspetti chiave:

- Il numero e le dimensioni degli strati convoluzionali.
- Il numero e le dimensioni degli strati fully-connected.
- Il tempo di addestramento.
- L'accuracy finale del modello.

Per raggiungere questo obiettivo, verrà utilizzato il dataset Keras CIFAR-10, che consiste in 60.000 immagini a colori di dimensioni 32x32 pixel, suddivise in 10 classi. Questo dataset rappresenta una sfida significativa per il riconoscimento di immagini, offrendo una base solida per valutare le performance della rete neurale in diverse configurazioni.

2 Struttura del progetto

Il progetto si articola su tre script Python separati:

- individual.py, script con la classe Individual;
- population.py, script con la classe Population;
- model.py, script per la creazione e valutazione del modello.

2.1 Individual

La classe Individual è composta come segue:

- Il costruttore:

```
    def __init__(self, dna1=None, dna2=None):

        self.dna = [None] * 4 #[n_conv, dim_conv, n_dense,
          ↪  dim_dense]
        self.accuracy = 0
        self.learning_time = 0
        self.fitness = 0.   #fitness calcolata come:
          ↪  accuracy/log10(learning_time)
        self.norm_fitness = 0 #fitness normalizzata sulla
          ↪  generazione

        if dna1 != None and dna2 != None:
            #caso di crossover
            cross = random.randint(1, len(self.dna)-1)
            self.dna = dna1[:cross] + dna2[cross:]
        elif dna1 != None:
            #copia del dna padre
            self.dna = dna1.copy()
        else:
            self.randomize()
```

Gestisce i casi in cui l'individuo debba essere un incrocio tra il dna di due individui, debba essere la copia di un individuo oppure deve essere randomizzato.

- Funzione di randomizzazione:

```
        def randomize(self):
        self.dna[0] = random.randint(2, 10) #n strati convoluzionali
        self.dna[1] = random.randint(8, 32) #dim strati
          ↪  convoluzionali
        self.dna[2] = random.randint(1, 5) #n strati densi
        self.dna[3] = random.randint(16, 320) #dim strati densi
```

Questa funzione randomizza casualmente, entro i parametri forniti, il dna dell'individuo.

- Funzioni per impostare le variabili dell'individuo:

```
1         def set_accuracy(self, accuracy):
2         self.accuracy = accuracy
3
4         def set_time(self, time):
5             self.learning_time = time
6
7         def evaluate(self):
8             self.fitness = self.accuracy /
                ↪  math.log10(self.learning_time)
```

- Le funzioni per la scrittura su file sono 3 e sono:
  - write_on_file_gene, scrive la generazione e il dna;
  - write_on_file_result, scrive la generazione, il dna, il tempo di addestramento, l'accuracy e la fitness;
  - write_on_file_result_norm, scrive la generazione, il dna, il tempo di addestramento, l'accuracy, la fitness e la fitness normalizzata.

2.2 Population

La classe Population è composta come segue:

- Una variabile "mutation_rate" che è una variabile float che imposta il rateo di mutazione di un individuo nella popolazione;

- Il costruttore:

```
        def __init__(self, size, file=None):
        self.size = size
        self.individuals = []
        self.selected = []
        self.offspring = []
        self.generation = 0
        if file is None:
            #se il file è None, creo una popolazione di individui casuali
            for i in range(size):
                self.individuals.append(Individual())
        else:
            with open(file, 'r') as f:
                righe = csv.DictReader(f)
                #salto l'header
                next(righe)

                #creo una lista delle righe e la inverto per recuperare gli ultimi individui
                lista_righe = list(righe)
                lista_righe = lista_righe[::-1]

                #recupero la generazione dell'ultimo individuo addestrato
                self.generation = lista_righe[0]["generazione"]

                #se la lunghezza delle righe è minore della size dichiarata, alcuni sono presi dal file e altri sono creati randomicamente
                if len(lista_righe)<size:
                    for i in range(len(lista_righe)):
                        #ast.literal_eval serve a trasformare il dna da stringa a lista
                        self.individuals.append(Individual
                    (ast.literal_eval(lista_righe[i]["dna"])))
                    for j in range(size-len(lista_righe)):
                        self.individuals.append(Individual())
                else:
                    for i in range(size):
                    self.individuals.append(Individual
                    (ast.literal_eval(lista_righe[i]["dna"])))
```

Questo costruttore tiene conto di tre possibilità:

- il file è "None" e la popolazione va creata da 0;
- il file è specificato ma le righe sono meno della size specificata e quindi parte della popolazione deve essere creata;
- la popolazione intera viene presa dal file.

- Funzione per la selezione degli individui:

```
        #metodo per selezionare gli individui su cui effettuare il
          ↪ crossover
    def select(self):

        #normalizzo la fitness degli individui
        self.normalize_fitness()
        for i in self.individuals:
            i.write_on_file_result_norm
            ("individui_result_norm.csv", self.generation)
        selecting=True
        self.selected.clear()

        #fino a quando la lunghezza della lista "selected" è minore
          ↪ della size, seleziono gli individui
        while selecting:
            for i in self.individuals:
                if random.random() < i.norm_fitness :
                    self.selected.append(i)
                if len(self.selected) == self.size:
                    selecting=False
                    break
```

Questa funzione prima normalizza le fitness degli individui della generazione usando una normalizzazione min-max, poi seleziona gli individui con la fitness normalizzata maggiore di un valore random. Questo ciclo si ferma quando la lunghezza della lista "selected" è uguale alla size della popolazione.

- Funzione di crossover:

```
        #metodo per effettuare il crossover
    def crossover(self):
        self.offspring.clear()
        for i in range(self.size):
            #seleziono due individui random e creo un terzo
              ↪ individuo il cui dna sarà un misto dei due
              ↪ genitori
            i1 = random.randint(0,self.size-1)
            i2 = random.randint(0,self.size-1)
            self.offspring.append(Individual(self.selected[i1].dna,
              ↪ self.selected[i2].dna))
```

Questa funzione prende randomicamente due individui e chiama il costrutture della classe Individual passando come argomento i dna degli individui scelti. Come sappiamo, quando vengono vengono passati due dna al costruttore di Individual, andrà a creare un terzo individuo il cui dna sarà una combinazione casuale di quelli passati come argomento.

- Funzione di mutazione:

```
        #metodo per la mutazione casuale
    def mutate(self):
        for i in self.offspring:
            if random.random() < self.mutation_rate:
                mutate_gene = random.randint(0, 3)
                if mutate_gene == 0:
                    i.dna[0] = random.randint(2, 10)
                if mutate_gene == 1:
                    i.dna[1] = random.randint(8, 32)
                if mutate_gene == 2:
                    i.dna[2] = random.randint(1, 5)
                if mutate_gene == 3:
                    i.dna[3] = random.randint(16, 320)
```

Questa funzione va a ciclare gli individui presenti nella lista "offspring", ovvero la nuova generazione, e, per ogni elemento della lista, confronta un numero random con il parametro "mutation_rate". Se il numero random è minore, si genera un numero casuale tra 0 e 3, ovvero si seleziona un indice casuale, e si modifica il gene all'indice selezionato.

- Funzione per normalizzare la fitness degli individui:

```
        #metodo per normalizzare la fitness degli indivisui
    def normalize_fitness(self):
        min_fit = 1
        max_fit = 0
        for i in self.individuals:
            if i.fitness < min_fit:
                min_fit = i.fitness
            elif i.fitness > max_fit:
                max_fit = i.fitness
        for i in self.individuals:
            #normalizzazione min-max
            i.norm_fitness = (i.fitness - min_fit) / (max_fit -
              ↪  min_fit)
```

Questa funzione recupera il valore minimo e massimo di fitness della generazione e, per ogni individuo, imposta il parametro "norm_fitness", che sarà la fitness in relazione agli altri individui della stessa generazione. Quindi l'individuo migliore avrà fitness 1, mentre il peggiore avrà 0.

- Inoltre sono presenti le funzioni:
  - "getBestIndividual": serve a recuperare l'individuo con la fitness più alta;
  - "add_generation": incrementa il contatore delle generazioni di 1.

2.3 Model

Questo script serve per creare i modelli, sulla base dei dna della e popolazione, addestrarli e valutarli. Inizialmente si scarica il dataset e si prepara per usarlo per l'addestramento e la valutazione.

```
#scarico il dataset
(train_images, train_labels), (test_images, test_labels) =
    ↪  datasets.cifar10.load_data()

#dichiaro le label del dataset
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

#converto le immagini in float
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')

#normalizzazione dell'immagine
train_images = train_images / 255
test_images = test_images / 255

#one-hot encoding per le label
num_classes = 10
train_labels = to_categorical(train_labels, num_classes)
test_labels = to_categorical(test_labels, num_classes)
```

Successivamente si dichiara: un "filename", se la popolazione deve essere recuperata da un file, e si istanzia una variabile dell'oggetto "Population". Infine si inizia il ciclo, che è il cuore vero e proprio dell'algoritmo genetico.

Il corpo del ciclo viene eseguito fin quando la variabile "running" ha valore "True". Il corpo del ciclo è strutturato come segue:

- Se la generazione della popolazione è 0 si incrementa in modo che la prima generazione sia 1;
- Se la variabile "writing" non è "False" si scrivono i dna dell'intera generazione su un file .csv. Questo serve principalmente per evitare che, se la popolazione è presa da file, ci siano doppioni all'interno del file contenente i dna fino ad ora generati.
- Successivamente si iterano tutti gli individui della popolazione. Per ogni individuo si crea il modello corrispondente al proprio dna e poi lo si addestra. Si recuperano l'accuracy sul test set e il tempo di addestramento e si assegnano all'individuo.
  Infine si calcola la fitness e si scrive l'individuo in un file separato.
- Alla fine del ciclo sugli individui si richiamano in ordine i metodi di Population:
  - Select;

- Crossover;
- Mutate.

Poi si sovrascrive "individuals" con la nuova generazione di individui "offspring" e si recupera il miglior individuo della popolazione. Se la sua fitness supera un certo valore, si imposta la variabile "running" a "False" per interrompere l'esecuzione.

```
        while running:
    #se la generazione è 0 (quindi se è stata appena creata), la
      ↪ aumento a 1
    if pop.generation == 0:
        pop.add_generation()

    #se writing è False, vuol dire che si sta leggendo da file e non
      ↪ si riscrivono gli individui
    if writing is not False:
        #scrivo i geni della generazione sul file, utile nel caso si
          ↪ debba fermare l'addestramento per non perdere i geni
        for individual in pop.individuals:
            individual.write_on_file_gene("individui_gene.csv",
              ↪ pop.generation)
    writing = True

    #ciclo per l'addestramento e valutazione dell'individuo
    for individual in pop.individuals:
        #clear_session serve a rilasciare lo stato globale di keras,
          ↪ serve quando si creano modelli in loop
        keras.backend.clear_session()

        #dichiarazione del modello
        model = Sequential()

        #aggiunta degli strati in base al dna dell'individuo
        create_model(individual.dna[0], individual.dna[1],
          ↪ individual.dna[2], individual.dna[3], model)

        #addestramento e valutazione del modello
        accuracy, training_time = training_and_evaluate(model)

        #setto le variabili dell'individuo e calcolo la fitness
        individual.set_accuracy(accuracy)
        individual.set_time(training_time)
        individual.evaluate()

        #scrivo le caratteristiche dell'individuo su file

          ↪ individual.write_on_file_result("individui_gene_result.csv",
          ↪ pop.generation)

        #libero per quanto possibile la memoria
        del model
        keras.backend.clear_session()
        gc.collect()

    #azioni di selezione, crossover e mutazione della popolazione
    pop.select()
    pop.crossover()
    pop.mutate()

    #sostituisco la vecchia generazione con quella nuova
    pop.individuals = pop.offspring

    #se un individuo ha superato la fitness soglia fermo
      ↪ l'esecuzione
    if pop.getBestIndividual().fitness >= 0.32:
        running = False

    #incremento la generazione
    pop.add_generation()
```

2.3.1 CNN

Nello script "model.py" sono presenti due funzioni:

- "create_model", che crea il modello in base al gene passato come argomento;
- "training_and_evaluate", compila il modello, lo addestra e lo valuta sul test set. Ritorna la precisione sul test set e il tempo di addestramento.

La rete neurale presa in considerazione ha uno schema fisso e viene generato come segue:

```
    #funzione per la creazione del modello
def create_model(n_conv, dim_conv, n_dense, dim_dense, model):
    for i in range(1, n_conv+1):
        if i == 1:
            #il primo strato convoluzionale ha l'argomento input_shape
            model.add(layers.Conv2D(dim_conv, (3,3), padding='same',
              ↪  activation='relu', input_shape=(32,32,3)))
        else:
            #dal secondo strato in poi aumento la dimensione dello strato
              ↪  convoluzionale
            model.add(layers.Conv2D(dim_conv*i, (3,3), padding='same',
              ↪  activation='relu'))

        #dopo ogni strato convoluzionale aggiungo uno strato
          ↪  BatchNormalization
        model.add(layers.BatchNormalization())

        #ogni due strati convoluzionali aggiungo uno strato di maxpooling
          ↪  e uno di dropout
        if i % 2 == 0:
            model.add(layers.MaxPooling2D(pool_size=(2,2)))
            model.add(layers.Dropout(0.5))

    #strato flatten prima degli strati densi
    model.add(layers.Flatten())

    for j in range(0, n_dense):
        model.add(layers.Dense(dim_dense, activation='relu'))
        model.add(layers.BatchNormalization())
        model.add(layers.Dropout(0.5))

    #strato denso di output
    model.add(layers.Dense(num_classes, activation='softmax'))
```

In sostanza si aggiungono gli strati convoluzionali, il primo avrà anche il parametro "input_shape", successivamente si aggiunge uno strato "BatchNormalization", ovvero uno strato che serve a normalizzare gli input. Ogni ciclo successivo al primo andrà ad aumentare la dimensione degli strati convoluzionali.
Ogni due strati convoluzionali si aggiungono uno strato di "MaxPooling" e uno di "Dropout" con un rateo di 0.5.

Dopo aver finito il ciclo che riguarda gli strati convoluzionali, si aggiunge uno strato "Flatten" per trasformare le matrici di output in un array da dare in input agli strati densi.
Nel ciclo per aggiungere gli strati densi, oltre ad aggiungere uno strato denso, si aggiungono anche uno strato di "BatchNormalization" e uno di "Dropout". Infine si aggiunge lo strato denso di output.

Nella funzione "training_and_evaluate" si compila, si addestra e si valuta il modello.

```
    #funzione per addestrare e valutare il modello
def training_and_evaluate(model):
    #compilazione del modello
    model.compile(optimizer='adam',
      ↪  loss=keras.losses.categorical_crossentropy,
      ↪  metrics=['accuracy'])

    #addestramento del modello e misurazione del tempo di addestramento
    start_time = time.time()
    history = model.fit(train_images, train_labels, epochs=7, verbose=2)
    end_time = time.time() - start_time

    #valutazione del modello sul test set
    results = model.evaluate(test_images, test_labels)
    print("test loss, test acc:", results)

    #ritorno della precisione misurata e del tempo di addestramento
    return results[1], end_time
```

Perciò si compila il modello con ottimizzatore "adam"e si effettua l'addestramento tramite il metodo "fit()". Per valutare l'accuratezza sul test set si usa il metodo "evaluate()" e infine si ritorna l'accuratezza e il tempo di addestramento.

3 Conclusioni

Dopo 11 generazioni è possibile notare come la popolazione converga naturalmente verso le soluzioni più "accettabili", ovvero quelle che riescono a restituire un'accuracy elevata nel minor tempo di addestramento possibile.
È possibile vedere come, dalla varibilità genetica elevata della prima generazione, si sia passati ad avere soluzioni simili dall'ottava generazione in poi. In particolare è possibile notare che:

- la scelta predominante nel numero di strati convoluzionali sia 7 o 10;
- la scelta predominante nella dimensione degli strati convoluzionali sia 19 o 30;
- la scelta predominante nel numero di strati densi e della loro dimensione sia 1 e 64.

Infatti i geni "migliori" e predominanti sono: [10, 30, 1, 64], [7, 19, 1, 64], [7, 30, 1, 64]. È quest'ultimo che ha registrato il valore di fitness più alto dell'intero algoritmo con 0.32, con una precisione di 0.82 e un tempo di addestramento di 356 secondi (circa 50 secondi a epoca).
Di seguito alcuni grafici che dimostrano l'andamento dell'algoritmo genetico in base al tempo di addestramento, all'accuracy e alla fitness per ogni generazione.

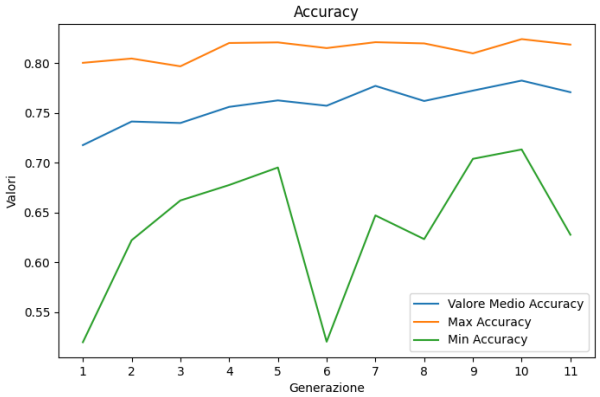

Figure 1: Andamento dell'accuracy nelle varie generazioni

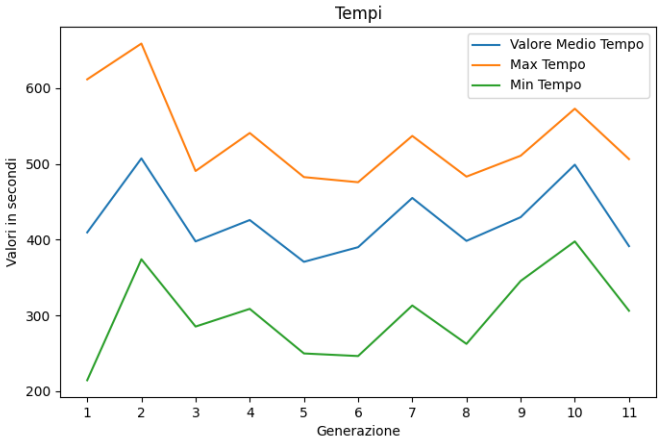

Figure 2: Andamento del tempo di addestramento nelle varie generazioni

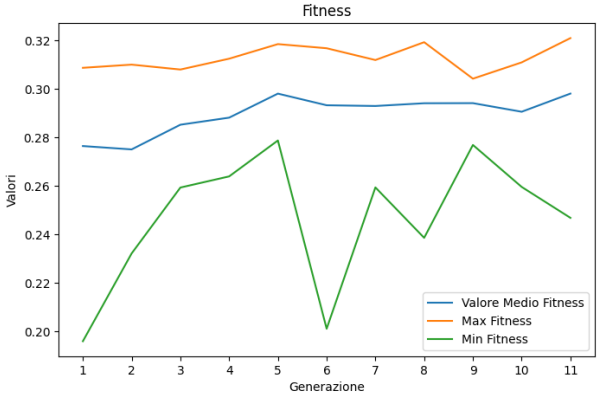

Figure 3: Andamento della fitness nelle varie generazioni

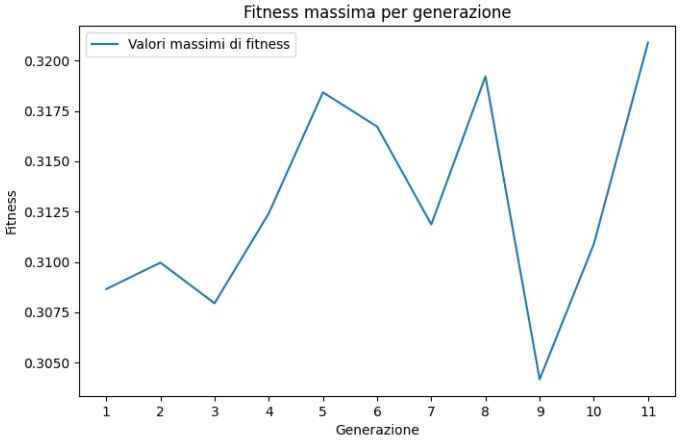

Figure 4: Focus sulle migliori fitness per ogni generazione

Inoltre è stato preso in considerazione il gene [7, 30, 1, 64] ed è stato effettuato un test con 100 epoche di addestramento. Alla fine dell'addestramento il modello è arrivato ad avere una precisione di circa 88%.

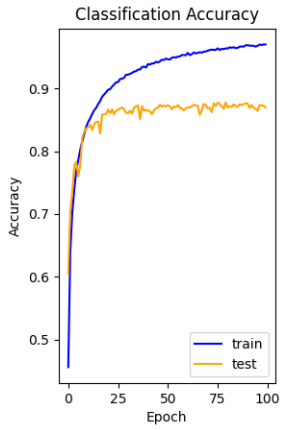

Figure 5: Andamento della precisione del modello su 100 epoche