CSE489 Selected Topics in Data Science – Natural Language Processing

Minor Task Documentation

Submitted to:

Prof. Dr. Alaa Hamdy

Submitted by:

Anthony Amgad Fayek 19P9880

Genetic Algorithm Implementation of Constant Area Coding (CAC) Compression

Table of Contents

1. Introduction

The efficiency of image compression algorithms is crucial in various applications, from reducing storage requirements to improving transmission speeds. One approach to image compression is Context-Adaptive Coding (CAC), a method that adapts the encoding process based on the content of the image to achieve higher compression rates. However, the performance of CAC significantly depends on the choice of kernel size used to process image chunks.

Identifying the optimal kernel size is a challenging task due to the vast search space and the dependency on the specific characteristics of the image dataset. A poorly chosen kernel size can lead to suboptimal compression, affecting both storage efficiency and image quality. The need for an effective method to determine the best kernel size is evident, and this is where Genetic Algorithms (GAs) come into play.

GAs are optimization techniques inspired by the principles of natural selection and genetics. They iteratively evolve a population of candidate solutions through processes such as selection, crossover, and mutation, gradually improving the solutions over successive generations. By mimicking the evolutionary process, GAs can efficiently search large and complex spaces to find optimal or near-optimal solutions.

In this project, we employ a GA to find the best kernel size for CAC compression on a given dataset of binary images. The goal is to maximize compression efficiency while maintaining image quality. By leveraging the adaptive and iterative nature of GAs, we aim to identify the optimal kernel size that balances the trade-off between compression rate and image fidelity, thereby enhancing the overall performance of CAC compression.

2. bintodec(binarr)

1. Purpose:
    - Converts a binary array to its decimal representation.
2. Parameters:
    - binarr (list of int): A list where each element is either 0 or 1, representing a binary number.
3. Returns:
    - num (int): The decimal equivalent of the binary array.
4. Internal Workings:

The function iterates through the binary array, converting each binary digit to its corresponding decimal value using the formula for binary to decimal conversion.

```
def bintodec(binarr):
    num = 0
    for i in range(len(binarr)):
        num += binarr[i] * (2**i)
    return num
```

3. class GA

The main class implementing the Genetic Algorithm.

3.1. __init__(self, path, population_size)

1. Purpose:
    - Initializes the GA object and sets up necessary variables.
2. Parameters:
    - path (str): Path to the dataset of images.
    - population_size (int): The size of the population for the genetic algorithm.
3. Internal Workings:
    - Initializes various attributes like min_x, min_y, original_pixel_size, x_gene_size, y_gene_size, and population.

- Calls find_min_size() to determine the smallest image dimensions and the total pixel size of all images.
- Calls generate_init_population() to create the initial population of candidate solutions.

```
class GA:
    def __init__(self, path, population_size) -> None:
        self.path = path
        self.min_x = float("inf")
        self.min_y = float("inf")
        self.original_pixel_size = 0
        self.x_gene_size = 0
        self.y_gene_size = 0
        self.population = []
        self.population_size = population_size
        self.find_min_size()
        self.generate_init_population()
```

3.2. find_min_size(self)

1. Purpose:
   - Finds the minimum dimensions (width and height) across all images in the dataset and calculates the total pixel size.
2. Returns:
   - self.min_x (int): The minimum width of the images.
   - self.min_y (int): The minimum height of the images.
3. Internal Workings:
   - Iterates over all images in the specified directory.
   - For each image, converts it to grayscale and updates min_x and min_y with the minimum dimensions found.
   - Accumulates the total pixel count in original_pixel_size.

```
def find_min_size(self):
        for p in os.listdir(self.path):
            if p.split('.')[-1] == 'gif':
                ret, img = cv2.VideoCapture(f"{self.path}/{p}").read()
                img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                self.min_x = min(img.shape[1], self.min_x)
                self.min_y = min(img.shape[0], self.min_y)
```

```
                self.original_pixel_size += (img.shape[0] * img.shape[1])
                del img
            elif p.split('.')[-1] == 'png':
                img = cv2.imread(f"{self.path}/{p}")
                img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                self.min_x = min(img.shape[1], self.min_x)
                self.min_y = min(img.shape[0], self.min_y)
                self.original_pixel_size += (img.shape[0] * img.shape[1])
                del img
```

3.3. generate_init_population(self)

1. Purpose:
    - Generates the initial population of binary arrays representing candidate kernel sizes.
2. Internal Workings:
    - Determines the gene size for x and y dimensions by calculating the ceiling of the log base 2 of min_x and min_y.
    - Generates random binary arrays of the combined size (x_gene_size + y_gene_size) for half of the specified population size.

```
def generate_init_population(self):
        self.x_gene_size = math.ceil(math.log2(self.min_x))
        self.y_gene_size = math.ceil(math.log2(self.min_y))
        for _ in range(int(self.population_size/2)):
            temp_arr = [random.randint(0,1) for _ in range(self.x_gene_size
+ self.y_gene_size)]
            self.population.append(temp_arr)
```

3.4. breeding(self)

1. Purpose:
    - Performs the breeding operation to generate offspring, including crossover and mutation.
2. Internal Workings:
    - Selects pairs of individuals from the population for crossover.
    - For each pair, a random crossover point is chosen, and two offspring are created by swapping segments of the parents' binary arrays at the crossover point.
    - Each offspring undergoes a random mutation at a randomly chosen index.

```
def breeding(self):
        for i in range(int(self.population_size/4)):
            crosspoint = random.randint(1,(self.x_gene_size +
self.y_gene_size - 1))
            restsize = (self.x_gene_size + self.y_gene_size) - crosspoint
            offspring1 = []
            offspring2 = []
            for j in range(crosspoint):
                offspring1.append(self.population[i][j])
                offspring2.append(self.population[i +
int(self.population_size/4)][j])
            for j in range(restsize):
                offspring1.append(self.population[i +
int(self.population_size/4)][crosspoint+j])
                offspring2.append(self.population[i][crosspoint+j])
            mutation_index = random.randint(0,(self.x_gene_size +
self.y_gene_size - 1))
            offspring1[mutation_index] = 1 - offspring1[mutation_index]
            self.population.append(offspring1)
            mutation_index = random.randint(0,(self.x_gene_size +
self.y_gene_size - 1))
            offspring2[mutation_index] = 1 - offspring2[mutation_index]
            self.population.append(offspring2)
```

3.5. CAC(self, img: np.array, kernelx, kernely)

1. Purpose:
    - Calculates the Context-Adaptive Coding (CAC) cost of an image given a kernel size.
    - This doesn't necessarily perform the compression in full. It only calculates the number of pixels there would be after compression. This was done for the sake of efficiency in calculations.
2. Parameters:
    - img (np.array): The binary image array.
    - kernelx (int): The width of the kernel.
    - kernely (int): The height of the kernel.
3. Returns:
    - sum (int): The calculated CAC cost.

4. Internal Workings:

- Thresholds the image to binary.
- Divides the image into chunks of size kernelx by kernely.
- For each chunk, counts the occurrences of homogeneous (all 0s or all 1s) and mixed chunks.
- Computes a cost based on the count and size of mixed chunks, and the occurrence of homogeneous chunks

```
def CAC(self, img: np.array, kernelx, kernely):
        ret, img = cv2.threshold(img, 50, 255, cv2.THRESH_BINARY)
        sum = 0
        zero_count = 0
        one_count = 0
        mix_count = 0
        mix_size = 0
        x_chunks = math.ceil(img.shape[1] / kernelx)
        y_chunks = math.ceil(img.shape[0] / kernely)
        for y_space in range(y_chunks):
            y_start = y_space * kernely
            y_end = (y_space+1) * kernely
            if y_space == (y_chunks-1):
                y_end = img.shape[0]
            for x_space in range(x_chunks):
                x_start = x_space * kernelx
                x_end = (x_space+1) * kernelx
                if x_space == (x_chunks-1):
                    x_end = img.shape[1]
                chunk = img[y_start:y_end, x_start:x_end]
                if len(np.unique(chunk)) == 2:
                    mix_count += 1
                    mix_size += (chunk.shape[0]*chunk.shape[1])
                else:
                    if np.unique(chunk)[0] == 0:
                        zero_count += 1
                    else:
                        one_count += 1
        counts = [zero_count, one_count, mix_count]
        max = np.argmax(counts)
        for i, count in enumerate(counts):
            if i == max:
                sum += count
            else:
                sum += (count*2)
        return sum + mix_size
```

3.6. decode(self, offspring)

1. Purpose:
    - Decodes a binary array (offspring) into kernel dimensions.
2. Parameters:
    - offspring (list of int): The binary array representing kernel dimensions.
3. Returns:
    - kernelx (int): The width of the kernel.
    - kernely (int): The height of the kernel.
4. Internal Workings:
    - Splits the binary array into two parts corresponding to x and y gene sizes.
    - Converts each part from binary to decimal using the bintodec function.

```
def decode(self, offspring):
        kernelxbin = offspring[:self.x_gene_size]
        kernelybin = offspring[self.x_gene_size:]
        kernelx = bintodec(kernelxbin)
        kernely = bintodec(kernelybin)
        return kernelx, kernely
```

3.7. fitness(self, offspring)

1. Purpose:
    - Calculates the fitness of an offspring based on its CAC cost across the dataset.
2. Parameters:
    - offspring (list of int): The binary array representing kernel dimensions.
3. Returns:
    - fitness (float): The fitness score of the offspring.
4. Internal Workings:
    - Decodes the binary array to kernel dimensions.
    - Checks for invalid kernel dimensions and assigns a fitness of 0 if found.

- Otherwise, computes the CAC cost for each image in the dataset using the specified kernel size.
- Returns the compression ratio between the original sum of the number of pixels and the sum of the CAC costs.

```
def fitness(self, offspring):
        kernelx, kernely = self.decode(offspring)
        if kernelx == 0 or kernely == 0 or kernelx > self.min_x or kernely
> self.min_y:
            return 0
        sum = 0
        for p in os.listdir(self.path):
            if p.split('.')[-1] == 'gif':
                ret, img = cv2.VideoCapture(f"{self.path}/{p}").read()
            elif p.split('.')[-1] == 'png':
                img = cv2.imread(f"{self.path}/{p}")
            else:
                continue
            img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            sum += self.CAC(img, kernelx, kernely)
            del img
        return self.original_pixel_size / sum
```

3.8. selection(self)

1. Purpose:
   - Selects the top-performing half of the population based on fitness scores.
2. Internal Workings:
   - Computes the fitness scores for all individuals in the population.
   - Sorts the population based on fitness scores.
   - Retains the top half of the population for the next generation.

```
def selection(self):
        fitness_scores = dict()
        for i, off in enumerate(self.population):
            fitness = self.fitness(off)
            fitness_scores.update({i:fitness})
        fitness_scores = dict(sorted(fitness_scores.items(), key=lambda
item: item[1]))
        mid = -1 * int(self.population_size/2)
        passed_off_index = list(fitness_scores.keys())[mid:]
        passed_off = []
        for i in passed_off_index:
```

```
            passed_off.append(self.population[i])
        self.population = passed_off
```

3.9. train(self, epochs)

1. Purpose:
    - Trains the genetic algorithm for a specified number of epochs.
2. Parameters:
    - epochs (int): The number of training epochs.
3. Internal Workings:
    - For each epoch, performs breeding and selection.
    - After each epoch, decodes and prints the best kernel dimensions found in the population.

```
def train(self, epochs):
        for _ in tqdm(range(epochs)):
            self.breeding()
            self.selection()
            x,y = self.decode(self.population[-1])
            print(f"\nBest Kernel this Epoch: x={x}, y={y}")
```

3.10. save_population(self, popstr)

1. Purpose:
    - Saves the current population to a file.
2. Parameters:
    - popstr (str): The filename to save the population.
3. Internal Workings:
    - Uses pickle to serialize and save the population to the specified file.

```
def save_population(self, popstr):
        pickle.dump(self.population, open(popstr,'wb'))
```

3.11. load_population(self, popstr)

1. Purpose:
    - Loads a population from a file.

2. Parameters:
    - popstr (str): The filename from which to load the population.
3. Internal Workings:
    - Uses pickle to deserialize and load the population from the specified file.

```
def load_population(self, popstr):
        self.population = pickle.load(open(popstr,'rb'))
```

4. Sample Run

Here's a sample Notebook on how the package should be used.

5. Important Links and Information

GitHub Repository

Here's a list of the packages required to run the Project:

- `numpy`
- `pickle`
- `opencv-python`
- `tqdm`