Sieć neuronowa uczona genetycznie

Filip Piskorski, Ewelina Badeja, Piotr Kądziela

Kraków 2023

Spis treści

Wstęp

Stworzona przez nas sieć nauronowa ma na celu rozpoznawanie obrazków znajdujących się w piętnastu kategoriach, takich jak rolki, łódź podwodna, kaktus, ognisko, itp. Została też dodana jedna kategoria, któa była przeznaczona na obrazki nierozpoznane. Projekt ma charakter eksperymentalny, ponieważ zamisat popularnych metod uczenia sieci, zastosowane zostało uczenie algorytmem genetycznym. Sieć neuronowa miała różne rozmiary w zależności od próby.
Sieć tworzona była przy użyciu samodzielnie napisanej na potrzeby projektu biblioteki.

Rozdział 1

Preliminaria

1.1 Standardowe podejście do rozpoznawania obrazków

Obecnie najlepszym sposobem na zidentyfikowanie obiektu na zdjęciu jest użycie sieci neuronowej, czyli programu, inspirowanego działaniem mózgu. Żeby napisać dobrą sieć neuronową, trzeba wykonać następunące kroki:

- **Zbieranie danych treningowych:** My korzystaliśmy z danych, zbieranych w ramach projektu Quick, Draw! (link: https://quickdraw.withgoogle.com/data).
- **Przygotowanie danych treningowych:** Niekiedy konieczne jest przetworzenie danych treningowych, aby były w odpowiednim formacie dla sieci neuronowej. Może to obejmować zmniejszenie rozmiaru obrazów, normalizację wartości pikseli czy też stosowanie innych technik przetwarzania obrazów.
- **Wybór architektury sieci neuronowej:** Jedną z popularnych architektur jest Convolutional Neural Network (CNN), która świetnie sprawdza się w zadaniach związanych z analizą obrazów.
- **Budowa modelu sieci neuronowej:** Przy implementacji sieci neuronowej przydatne są biblioteki, takie jak TensorFlow, Keras lub PyTorch, które dostarczają narzędzi i interfejsy programistyczne do tworzenia i trenowania modeli sieci neuronowych. My zdecydowaliśmy się ograniczyć korzystanie z funkcji bibliotecznych do minimum.
- **Trenowanie modelu:** Trenowanie sieci polega to na dostarczeniu danych treningowych do modelu i aktualizowaniu wag sieci neuronowej w celu minimalizacji funkcji kosztu. Proces trenowania wymaga odpowiedniej konfiguracji parametrów, takich jak tempo uczenia czy rozmiar partii (batch size).

- **Testowanie modelu:** Model należy testować na nowych, niewidzianych wcześniej danych, ocenić jego działanie i w razie potrzeby dostosować wybrane parametry.

1.2 Podejście genetyczne

Algorytmy genetyczne działają na podstawie modelu ewolucji populacji, gdzie rozwiązania potencjalne są reprezentowane jako osobniki w populacji. Wykorzystuje się je do rozwiązywania problemów optymalizacyjnych.

W naszym przypadku celem było zoptymalizowanie trafności rozpoznawania obrazków. W tym celu przy każdej iteracji (dla każdego nowego "pokolenia") były wykonywane następujące kroki:

- **Inicjalizacja populacji początkowej:** Na początku algorytmu tworzona jest losowo populacja początkowa, składająca się z pewnej liczby osobników. Każdy osobnik reprezentuje potencjalne rozwiązanie problemu i jest kodowany za pomocą genotypu, którym są w naszym przypadku wagi połączeń oraz funkcja aktywacji.

- **Ocena osobników:** Każdy osobnik w populacji jest oceniany na podstawie funkcji oceny. W naszym przypadku ocena była wystawiana na podstawie fitnessu, czyli sumy błędów, jakie popełniła dana sieć.

- **Selekcja:** Proces selekcji polega na wyborze osobników z populacji na podstawie ich wartości przystosowania. Osobniki, które nie zostały wybrane, zostają usunięte.

- **Krzyżowanie:** Wybrane osobniki są poddawane operacji krzyżowania (crossover). Krzyżowanie polega na wymianie fragmentów genotypów między dwoma osobnikami w celu wygenerowania potomstwa. Operacja ta ma na celu łączenie cech różnych rozwiązań, co może prowadzić do odkrycia nowych i lepszych sieci.

- **Mutacja:** Po operacji krzyżowania, część potomstwa jest poddawana mutacji. Mutacja polega na losowej zmianie pewnej liczby genów w genotypie potomstwa. Ma to na celu wprowadzenie różnorodności genetycznej do populacji i zapobieganie zatrzymaniu się algorytmu na lokalnych ekstremach. Prawdopodobieństwo mutacji jest zwykle niewielkie, aby uniknąć zbyt gwałtownych zmian.

- **Zastąpienie:** Po operacjach krzyżowania i mutacji potomkowie są włączani do populacji na miejsce części usuniętych osobników. Następnie generowane są nowe losowe sieci, przez co w każdym pokoleniu liczba ocenianych osobników pozostaje taka sama.

Rozdział 2

Dokumentacja kodu

2.1 Wstęp

W tym rozdziale zostaną przedstawione kluczowe klasy i funkcje biblioteczne, potrzebne do prawidłowego działania programu, wraz z przykładem ich użycia.

2.2 Klasa Layer

```
class Layer(object):
    activation_fn = None
    activation_fn_vectorized = None
    weights = None
    bias = None

    def __init__(self, num_units=None, weights=None, bias=None, activation_fn=lambda x: x, prev_layer=None,
                 init_from_parents=False):
        if init_from_parents:
            self.values = None
            self.prev_layer = prev_layer
            self.weights = None
            self.bias = None
            return
        if activation_fn is None and prev_layer is not None:
            raise ValueError("Activation function must be provided")
        if weights is None and prev_layer is not None:
            raise ValueError("Weights must be provided")
        if bias is None:
            bias = np.zeros(num_units)
        if prev_layer is not None and len(bias) != num_units:
            raise ValueError("Bias must be the same length as num_units")
        if prev_layer is not None and len(weights[0]) != num_units:
            raise ValueError("Weights and bias must be the same length")
        self.values = None
        self.prev_layer = prev_layer
        self.num_units = num_units
        if prev_layer is None:
            return
        self.weights = np.array(weights).copy()
        self.bias = np.array(bias).copy()
        self.activation_fn = activation_fn  # przerobić na np.vectorize
        self.activation_fn_vectorized = np.vectorize(activation_fn)
        return
```

Jest to klasa przechowująca dane o wartwie w sieci neuronowej. Przechowuje ona informacje o wagach jak i o bias'ie poszczególnych neuronów w postaci macierzy. Ma ona również dostęp do poprzedniej warstwy Wartości są zapisywane w polu values i aktualizowane za pomocą funkcji update_value.

```
def update_value(self, x=None):
    if x is None:
        if self.prev_layer is None:
            raise ValueError("No input provided")
        x = self.prev_layer.values
    elif len(x) != self.num_units:
        raise ValueError("Input size does not match layer size")
    self.values = self.activation_fn_vectorized((x @ self.weights) + self.bias)
    return
```

2.3 Klasa ActivationFunction

```
class ActivationFunction(Enum):
    def __call__(self, x):
        return self.value(x)

    def __str__(self):
        return self.value

    identity = identity_fn
    sigmoid = sigmoid_fn
    tanh = tanh_fn
    relu = relu_fn
    leaky_relu = leaky_relu_fn
    softmax = softmax_fn
```

Jest to klasa reprezentująca funkcję aktywacji. W konstruktorze podawana jest funkcja, ktorej chcemy używać jeśli wywołamy dany obiekt.

2.4 Klasa NeuralNetwork

```
class NeuralNetwork(object):
    def __init__(self, layer_count=None, layers_units_count=None, weights=None, biases=None, activation_fn=lambda x: x,
                 init_from_parents=False):
        if init_from_parents:
            self.layers = None
            self.layer_count = None
            self.layers_units_count = None
            self.activation_fn = None
            return
        if layer_count is None:
            raise ValueError("layer_count must be provided")
        if layers_units_count is None:
            raise ValueError("layers_units_count must be provided")
        if weights is None:
            raise ValueError("weights must be provided")
        if biases is None:
            raise ValueError("biases must be provided")
        if len(weights) != layer_count-1:
            raise ValueError("weights must be the same length as layer_count")
        if len(biases) != layer_count-1:
            raise ValueError("biases must be the same length as layer_count")
        if len(layers_units_count) != layer_count:
            raise ValueError("layers_units_count must be the same length as layer_count")
        self.layers = [Layer.Layer(layers_units_count[0], None, None, activation_fn, None, False)]
        for i in range(1, layer_count):
            self.layers.append(
                Layer.Layer(layers_units_count[i], weights[i-1], biases[i-1], activation_fn, self.layers[i - 1]))
        self.layer_count = layer_count
        self.layers_units_count = layers_units_count
        self.activation_fn = activation_fn
        return
```

Klasa ta jest reprezentacją sieci neuronowej. W konstruktorze podajemy wszystkie parametry dotyczace sieci. Do obliczania wartości output'u dla danego input'u przeznaczone są funkcje calculate_value i calculate_value_with_softmax. Różnią się one tym, że ta druga po obliczeniu wartości neuronów w ostatniej warstwie przepuszcza te wyiki przez funkcję softmax.

```
def calculate_value(self, x=None):
    if x is None:
        raise ValueError("No input provided")
    self.layers[0].set_value(x)
    for i in range(1, self.layer_count):
        self.layers[i].update_value()
    return self.layers[self.layer_count - 1].values
```

2.5 Klasa NeuralNetworkFactory

```
class NeuralNetworkFactory:
    def __init__(self):
        return

    @staticmethod
    def create_random_neural_network(layer_count, layers_units_count, activation_fn=None):
        if layer_count is None:
            raise ValueError("layer_count must be provided")
        if layers_units_count is None:
            raise ValueError("layers_units_count must be provided")
        if len(layers_units_count) != layer_count:
            raise ValueError("layers_units_count must be the same length as layer_count")
        weights = []
        biases = []
        if activation_fn is None:
            activation_fn = af.ActivationFunction(
                af.availible_activation_functions[random.randint(0, len(af.availible_activation_functions) - 1)])
        for i in range(1, layer_count):
            weights.append(np.random.normal(0, 1, (layers_units_count[i - 1], layers_units_count[i])))
            biases.append(np.random.normal(0, 1, layers_units_count[i]))
        return NeuralNetwork.NeuralNetwork(layer_count, layers_units_count, weights, biases, activation_fn)

    @staticmethod
    def create_neural_network_from_parents(parent1, parent2):
        if parent1 is None:
            raise ValueError("Parent1 must be provided")
        if parent2 is None:
            raise ValueError("Parent2 must be provided")
        if parent1.layer_count != parent2.layer_count:
            raise ValueError("Parents must have the same number of layers")
        if parent1.layers_units_count != parent2.layers_units_count:
            raise ValueError("Parents must have the same number of units in each layer")
        nn = NeuralNetwork.NeuralNetwork(init_from_parents=True)
        nn.init_from_parents(parent1, parent2)
        return nn
```

Ta klasa jest odpowiedzialna za tworzenie losowych sieci neuronowych, sieci neuronowych z rodziców, zapisywanie i ładowaie sieci neuronowych.

2.6 Klasa ModelDoodlesFiles

```
class ModelDoodlesFiles:
    def __init__(self, image_size=128):
        print("Loading np bitmaps...")
        self.model_doodles_categories = open("./categories.txt", "r").read(-1).split("\n")
        self.model_doodles_excluded_categories = open("./categories.txt", "r").read(-1).split("\n")
        for i in self.model_doodles_categories:
            self.model_doodles_excluded_categories.remove(i)
        self.model_doodles_samples = []
        for i in self.model_doodles_categories:
            print("Loading " + i + "...")
            self.model_doodles_samples.append(
                np.load("./npbitmaps" + str(image_size) + "x" + str(image_size) + "/full_numpy_bitmap_" + i + ".npy"))

            self.model_doodles_categories = self.model_doodles_categories[:10000]
        for i in self.model_doodles_excluded_categories:
            print("Loading " + i + "...")
            self.model_doodles_samples.append(
                np.load("./npbitmaps" + str(image_size) + "x" + str(image_size) + "/full_numpy_bitmap_" + i + ".npy"))
            self.model_doodles_categories = self.model_doodles_categories[:10000]
```

Aby nie ładować dla każdego modelu osobno plików do uczenia stworzyliśmy klasę ModelDoodlesFiles, która je przechowuje i w razie potrzeby udostępnia.

2.7 Klasa ModelDoodles

```
class ModelDoodles:
    def __init__(self, files, layers_count=None, layers_units_count=None, special_init=False, activation_fn=None,
                 side=128):
        if files is None:
            raise ValueError("files must be provided")
        self.files = files
        if special_init:
            return
        if layers_count is None:
            raise ValueError("layers_count must be provided")
        if layers_units_count is None:
            raise ValueError("layers_units_count must be provided")
        if len(layers_units_count) != layers_count:
            raise ValueError("layers_units_count must be the same length as layers_count")
        if layers_count < 2:
            raise ValueError("layers_count must be at least 2")
        if layers_units_count[0] != side * side:
            raise ValueError("layers_units_count[0] must be " + str(side * side))
        if layers_units_count[layers_count - 1] != len(self.files.model_doodles_categories) + 1:
            raise ValueError(
                "layers_units_count[layers_count-1] must be " + str(len(self.files.model_doodles_categories) + 1))
        self.side = side
        self.nn = nnf.NeuralNetworkFactory().create_random_neural_network(layers_count, layers_units_count,
                                                                          activation_fn=activation_fn)
        self.fitness = self.calculate_fitness()
        return
```

Klasa ModelDoodles jest odpowiedzialna za stworzenie klasy NeuralNetwork, uczenie jej, mutowanie i kojarzenie. Jest również odpowiedzialna za liczenie fitess'u danej sieci neuronowej i optymalizację tego procesu.

```
def calculate_fitness(self, samples_count=64):
    self.fitness = 0
    for i in range(0, samples_count):
        sample_set = random.randint(0, len(self.files.model_doodles_samples) - 1)
        sample = self.files.model_doodles_samples[sample_set][
            random.randint(0, self.files.model_doodles_samples[sample_set].shape[0] - 1)]
        result = self.nn.calculate_value(sample)
        result -= np.max(result)  # softmax
        result = np.exp(result)
        result = result / np.sum(result)
        if sample_set < len(self.files.model_doodles_categories):
            r_2 = result.copy()
            r_2[sample_set] = 1 - r_2[sample_set]
            np.square(r_2, r_2)
            self.fitness -= np.sum(r_2)
        else:
            r_2 = result.copy()
            r_2[len(self.files.model_doodles_categories)] = 1 - r_2[len(self.files.model_doodles_categories)]
            np.square(r_2, r_2)
            self.fitness -= np.sum(r_2)
        pass
    self.fitness /= samples_count
    return self.fitness
```

2.8 Klasa GeneticAlgorithmTrainer

```
class GeneticAlgorithmTrainer:
    def __init__(self, population_size=120, layers_count=5, layers_units_count=None, side=32, activation_fn=None):
        if layers_units_count is None:
            layers_units_count = [32 * 32, 1000, 500, 400, 16]
        self.files = ModelDoodlesFiles.ModelDoodlesFiles(image_size=side)
        self.population_size = population_size
        self.layers_count = layers_count
        self.layers_units_count = layers_units_count
        self.population = []
        self.best_ever_nn = None
        self. best_ever_fitness = float("-inf")
        for i in range(0, population_size):
            self.population.append(ModelDoodles.ModelDoodles(self.files, layers_count, layers_units_count, False,
                                                             side=side, activation_fn=activation_fn))
        return
```

Klasa GeneticAlgorithmTrainer jest odpowiedzialna za tworzenie modeli i naukę ich. Nauka odbywa się przez wywołanie funkcji iterate.

```
def iterate(self, mutation_rate=0.0001):
    # print("Sorting...")
    for i in range(0, len(self.population)):
        # print("Calculating fitness of doodle " + str(i) + " of " + str(len(self.population)))
        self.population[i].get_fitness()
    self.population.sort(key=lambda x: x.get_fitness())
    # print("Cutting population...")
    self.population = self.population[int(self.population_size / 3):]
    # print("New population size: " + str(len(self.population)))
    # print("Creating new population...")
    random.shuffle(self.population)
    for i in range(0, int(self.population_size / 3)):
        self.population.append(
            ModelDoodles.ModelDoodles(self.files, self.layers_count, self.layers_units_count, True))
        self.population[len(self.population) - 1].init_from_parents(self.population[2 * i],
                                                                    self.population[2 * i + 1])
    # print("New population size: " + str(len(self.population)))
    # print("Mutating...")
    for i in range(0, len(self.population)):
        # print("Mutating doodle " + str(i) + " of " + str(len(self.population)))
        self.population[i].mutate(mutation_rate)
    # print("Sorting...")
    for i in range(0, len(self.population)):
        # print("Calculating fitness of doodle " + str(i) + " of " + str(len(self.population)))
        self.population[i].get_fitness()
    self.population.sort(key=lambda x: x.get_fitness())
    if self.best_ever_fitness <= self.population[len(self.population) - 1].get_fitness():
        self.best_ever_fitness = self.population[len(self.population) - 1].get_fitness()
        self.best_ever_nn = NeuralNetwork(self.layers_count, self.layers_units_count,
                                          self.population[len(self.population) - 1].get_weights(),
                                          self.population[len(self.population) - 1].get_biases(),
                                          self.population[len(self.population) - 1].nn.activation_fn)
    return
```

2.9 Przykład użycia neszej biblioteki

Klasy opisane w punktach 2.2 - 2.8 Typowe użycie biblioteki wygląda w ten sposób:

```
if __name__ == '__main__':
    print("Initializing...")
    population = ga.GeneticAlgorithmTrainer(600, 4, [32*32, 50, 20,  16], side=32,
                                            activation_fn=ActivationFunction.ActivationFunction.tanh)
    for i in range(0, 10000000):
        print("Iteration " + str(i) + " starting...")
        start_time = time.time()
        population.iterate(0.00000000001)
        print("Iteration " + str(i) + " fitness: " + str(population.get_best().get_fitness()) + " time: " +
              str(time.time() - start_time))
        print("Average fitness: " + str(np.average([x.get_fitness() for x in population.population])))
        print("Best ever fitness: " + str(population.get_best_ever_fitness()))
        NeuralNetworkFactory().save_neural_network_to_file(population.get_best_ever(), "./best_neural_network.picle")
    print(population.population[0].get_fitness())
    population.population[0].fitness = None
    exit(0)
```

Na początku programu tworzymy trenera genetycznego, z odpowiednią wielkością populacji i wielkością sieci. Następnie program iteruje w pętli for funkcją population.iterate(). Program po każdej iteracji zapisuje najlepszą sieć do pliku.

Rozdział 3

Napotkane problemy

3.1 Wstęp

W tym rozdziale zostaną opisane problemy, z którymi musieliśmy się zmierzyć oraz jakie kroki podjęliśmy, żeby je rozwiązać.

3.2 Problem z kategorią “nie rozpoznano”

- **Problem:** Program zdał sobie sprawę, że opłaca się wrzucać większośc obrazków do kategorii “nie rozpoznano”.
- **Próba rozwiązania problemu:** Uczenie algorytmu tylko na obrazkach z wybranych piętnastu kategorii, które miał rozpoznawać.
- **Skutek:** Program rzadziej wybierał kategorię “nie rozpoznano”, jednak dalej nie odgadywał kategorii z pożądaną skutecznością.

3.3 Próba zmiany rozdzielczości

- **Problem:** Program uczył się zbyt wolno.
- **Próba rozwiązania problemu:** Zmniejszenie rozdzielczości obrazków z 128x128 na 32x32 i 28x28.
- **Skutek:** Znaczna część obrazków stała się nieczytelna (nie można było winić programu za to, że nie wie co jest na obrazku, bo sami nie byliśmy w stanie tego odgadnąć).

3.4 Zmiana liczby kategorii

- **Problem:** Program uczył się zbyt wolno.
- **Próba rozwiązania problemu:** Zmniejszenie liczby kategorii.
- **Skutek:** Nawet dla dwóch kategorii program mylił się zbyt często.

Rozdział 4

Wnioski

4.1 Nasze przemyślenia

Sieć neuronowa uczona genetycznie nie nadaje się do rozpoznawania obrazków. Uczyła się ona zdecydowanie zdbyt wolno, by można uznać to podejście za efektywne. Ciekawym zjawiskiem, które udało się zaobserwować, było to,że nasz program nie działał poprawnie, jednak wykazywał się swego rodzaju “inteligencją”. Dla przykładu często znajdował rozwiązania, które statystycznie najbardziej opłacało się wybierać (tak było np. z kategorią ”nie rozpoznano”, przed poprawieniem sposobu uczenia).

4.2 Zastosowania wybranego podejścia

Nie jesteśmy pierwszym zespołem, który wpadł na pomysł sieci neuronowej uczonej genetycznie. Wcześniej algorytmy tego typu były wykorzystywane m. in. do balansowania kijka na wózku. Wspólną cechą większości sieci, które skutecznie uczyły się podejściem biologicznym, była mniejsza liczba danych wejściowych (oraz co za tym idzie mniejsza liczba neuronów). Liczba ta musi być rzędu kilka lub kilkanaście, a w naszym przypadku samych danych wejściowych było co najmniej 748. Spora zaleta podejścia genetycznego została opisana w Leksykonie Sieci Neuronowych. Cytując: ”Podstawowa zaleta stosowania GNN jest fakt, z e funkcja przystosowania ocenia jakosc nieliniowych modeli neuronowych i dlatego nie zachodzi niebezpieczeństwo wyeliminowania w czasie procedury optymalizacyjnej jakiegos kluczowego nieliniowego zwiazku. Ponadto metoda dostarcza zestawu dobrze działajacych sieci, ktore sa w stanie rozwiazac dany problem często na rozne, komplementarne sposoby.”[1)]

[1)] R. Tadusiewicz, M. Szaleniec. *Leksykon Sieci Neuronowych*. 2015

4.3 Końcowe wnioski

Genetyczne uczenie sieci neuronowych jest procesem powolnym i sprawdza się tylko dla mniejszych sieci. Jego zaletą jest jednak to, że można w ten sposób znaleźć wiele alterantywnych, działających poprawnie sieci.