Marcin Zięba

Inteligencja obliczeniowa

Raport 6

Temat: **Implementacja algorytmów ewolucyjnych na GPU z użyciem TensorFlow**

TL;DR; Dopracowałem implementację algorytmu EMAS, uwzględniając większość uprzednio uzgodnionych wymagań, co do implementacji - poza wyspami. Aktualna wersja kodu kompiluje się przez XLA, umożliwiając uruchamianie na TPU oraz przyśpieszając wykonanie na pozostałych platformach obliczeniowych. Zostały dodane nowe operacje krzyżowania oraz zliczanie operacji krzyżowania. Co nieco zaskakujące, uruchomienie na TPU nie okazało się dużo szybsze od CPU.
Uwaga: Podczas oddawania projektu został wykryty błąd w implementacji. Poprawka została opisana w dodatkowej części pod koniec raportu.

Colab: [Final EMAS colab notebook](https://colab.research.google.com/drive/1t8Dfwt54JK1_qITwiKgxzYu4g8L5GDT5?usp=sharing)
https://colab.research.google.com/drive/1t8Dfwt54JK1_qITwiKgxzYu4g8L5GDT5?usp=sharing

Q&A

Co zostało zaimplementowane?

Napisana poprzednio podstawowa wersja algorytmu EMAS (Evolutionary Multi-Agent System) została dopracowana. Uzgodnione uprzednio zmiany dotyczyły następujących elementów:

- dostosować kod tak, aby był kompilowalny przez XLA,
- dodać do implementacji wyspy oraz mechanizm migracji agentów między nimi,
- zrobić zliczanie operacji krzyżowania,
- dodać nowe operacje krzyżowania.

Punkty zaznaczone kolorem zielonym zostały poprawnie wykonane, natomiast ten na czerwono - nie.

Co zostało zrobione w ramach umożliwienia kompilacji przez XLA?

Głównym problemem była zmienna długość tensorów w kolejnych iteracjach pętli while. Aby poradzić sobie z nim zostało zastosowane wykorzystanie pętli while w formie explicite oraz dopełnianie problematycznych tensorów do prespecyfikowanej długości wyznaczonej przez maksymalną dozwoloną wielkość populacji. Ponadto operacja shuffle wykorzystana do mieszania populacji została zastąpiona prostszą wersją opartą o resztę z dzielenia.

```
perm = (tf.cast(tf.range(current_pop_size), dtype=tf.int32) * 24179) % current_pop_size
energy = tf.gather(energy[:current_pop_size], perm)
```

Jak zostały dodane wyspy oraz mechanizm migracji?

Nie zostały dodane. Okazało się, że dostosowanie kodu pod XLA zajęło mi więcej czasu niż się spodziewałem, sprawiając, że byłem już zbyt zdemotywowany, żeby na poważnie przysiąść do tego punktu.

Jakie operacje krzyżowania zostały dodane?

```python
@tf.function()
def crossing(a, b, n):
    k = tf.reduce_sum(tf.ones_like(a[:, 0], dtype=tf.int32))

    # Random bubble (hyper-sphere) between points
    reposition = tf.random.uniform([k, n], minval=-1, maxval=1)
    reposition = reposition / tf.norm(reposition, axis=1, keepdims=True)
    new_position_1 = ((a + b) / 2.0) + (tf.maximum((tf.norm(a - b, axis=1, keepdims=True) / 2.0), 1e-3) * reposition)

    # Random exchange of genomes
    new_position_2 = tf.where(
        tf.random.uniform(shape=[k, 1]) < tf.random.uniform([1], 0.05, 0.95), a, b)

    # Choose one crossing or another
    new_position = tf.where(
        tf.random.uniform(shape=[k, 1]) < tf.random.uniform([1], 0.2, 0.5), new_position_1, new_position_2)

    # Apply mutation, because why not
    # Cauchy distribution derived from uniform distribution
    mutation = 0.05 * tf.math.tan(tf.random.uniform(minval=-1.0 * math.pi / 2 + 1e-6, maxval=math.pi / 2 - 1e-6, shape=[k, n]))
    new_position_m = tf.where(
        tf.random.uniform(shape=[k, 1]) < tf.random.uniform([1], 0.05, 0.4), new_position + mutation, new_position)

    # Choose mutated or not
    new_position = tf.where(
        tf.random.uniform(shape=[k, 1]) < 0.4, new_position_m, new_position)

    new_position = tf.clip_by_value(new_position, clip_value_min=-5.12, clip_value_max=5.12)

    return new_position
```

1. Losowanie punktów z wnętrza hiperkuli będącej pomiędzy obiema agentami
2. Wybór elementu genomu losowo z jednego bądź drugiego agenta
3. Mutacja losowana z rozkładu Cauchy'ego

Każda z powyższych operacji ma oczywiście pewną, raczej rozsądną (choć to pewnie hiper-parametr, którym warto by się zająć), szansę zajścia.

Jaką funkcję optymalizowałem?

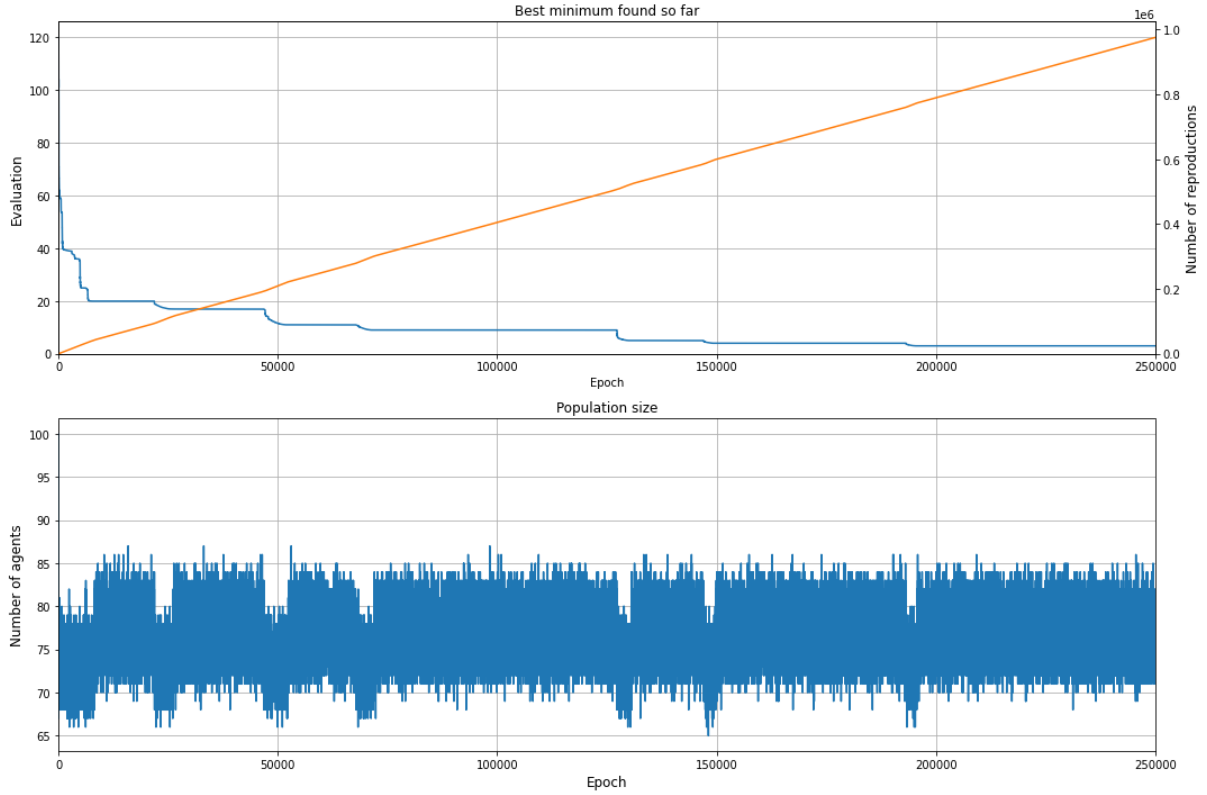
Funkcja Rastrigina dla n-wymiarów. Testowałem w wersji lekkiej 10-D, ale długiej (250,000 epok), oraz cięższej 10000-D, ale krótszej (100 epok).

Jakie hiper-parametry symulacji przyjąłem?

Liczba agentów: 100
Początkowa energia: 5 dla każdego agenta
Zmiana energii po walce: 2 (przegrany -2, wygrany +2)
Energia wymagana do preprodukcji: 10
Koszt reprodukcji: 2.5 (obu agentów -2.5, nowy agent startuje z 5)

Czy implementacja okazała się poprawna i zbiegała?

Wygląda na to, że tak. Poniżej przedstawione są statystyki uruchomienia dla 100 agentów w populacji, dla 10-wymiarowej funkcji Rastrigina w obrębie 250000 epok.



Pierwszy wykres przedstawia zmianę najlepszego rozwiązania w czasie oraz liczbę krzyżowań. Ewaluacje są wykonywane tylko w obrębie areny walk.
Drugi wykres przedstawia zmianę rozmiaru populacji w czasie. Jak widać, dla zadanych hiper-parametrów populacja bardzo szybko osiąga stabilny rozmiar około 70-85 osobników, oraz widać momenty, w których w puli genetycznej pojawiły się interesujące mutacje.

Dla pewności, suma energii w systemie jest kontrolowana przy każdym uruchomieniu na sam koniec procesu ewolucyjnego.

```
assert tf.abs(tf.reduce_sum(energy) - k * 5) < 1e-4, "Energy not preserved"
```

Jak wygląda kwestia czasu obliczeń?

Rozmiar problemu: niewielki, ale długi

- liczba agentów: 100
- liczba wymiarów: 10
- liczba epok: 250,000

Rozmiar problemu: duży, ale krótki

- liczba agentów: 2000
- liczba wymiarów: 10000
- liczba epok: 100

Przez “bez XLA” w poniższej tabelce rozumiemy, że zostały wykorzystane wszystkie optymalizacje poza kompilacją XLA (jit_compile).

	Niewielki, ale długi		Duży, ale krótki	
	bez XLA	XLA	bez XLA	XLA
CPU	14.9583s	12.1042s	64.3765s	60.5606s
GPU - Standard	43.2964s	38.9117s	0.5612s	0.4856s
GPU - Premium	52.3314s	50.5861s	0.1559s	0.1379s
TPU	niewykonalne	11.5694s	niewykonalne	58.9382s

Ogromnym zaskoczeniem jest wynik dla TPU, który jest bardzo zbliżony do czasów dla zwykłego CPU. Ciężko określić czy obliczenia były faktycznie wykonywane na TPU (powinny), czy też w tym wypadku z jakiegoś powodu mimo wszystko wykonało się na CPU. Jeśli faktycznie wykonało się to na TPU, to możliwe, że kompilator postanowił zlinearyzować kod na wzór tego dla CPU, kompletnie nie wykorzystując potencjału do akceleracji - notabene czasy były powtarzalnie lepsze (chociaż tylko odrobinkę) od CPU, ale nie mam tego nigdzie formalnie udowodnione dla większej próby, więc to mógł być też przypadek.

W ogólności natomiast widać, że po kompilacji XLA kod na wszystkich maszynach jest faktycznie nieznacznie szybszy od kodu, który miał wszystkie optymalizacje poza tą jedną.

Wnioski z projektu

Pisanie kodu kompilowalnego do XLA w TensorFlow, o ile raczej łatwiejsze i bardziej wysokopoziomowe niż pisanie w czystej Cudzie, nadal pozostaje sporym wyzwaniem, gdyż pojawia się dużo nieoczywistym problemów:

- niektóre operacje nie są wspierane, trzeba mieć świadomość wielu ograniczeń,
- debugowanie jakiegokolwiek dłuższego kodu jest wyzwaniem samym w sobie:
    - wiele błędów ujawnia się dopiero, gdy dodana zostanie flaga uruchamiająca kompilację kodu, bez której może działać idealnie,
    - na stosie błędów wywołania są do funkcji ze zbudowanego grafu obliczeniowego, a nie oryginalnego kodu.

Mimo to, z odpowiednią dozą samozaparcia i kreatywności możliwe jest wykonanie dosyć skomplikowanych operacji - w ramach projektu były to: wcześniej algorytm mrówkowy, oraz teraz algorytm EMAS.

Pod względem szybkości działania takiego kodu, trzeba zaznaczyć, że testy wykonywane w trakcie projektu pokazywały przyśpieszenia na poziomie nawet dwóch rzędów wielkości, względem czystego pythona wywołującego operacje interaktywnie operacje tensorflow. Widoczne jest również przyśpieszenie względem funkcji, dla których wyznaczony zostaje graf obliczeń (flaga tf.function).

Korekta po oddaniu projektu

Na koniec oddawania projektu zauważyłem potencjalny błąd w implementacji. Błąd faktycznie był, a jego usunięcie znacząco poprawiło zbieżność algorytmu. Brakowało jednego wymiaru w dwóch wywołaniach generatora liczb losowych w funkcji dokonującej krzyżowania, przez co operacje genetyczne w efekcie nie były wykonywane w zamierzony sposób.

Nie zostały dodatkowo dopasowywane żadne hiper-parametry - wyłącznie ten jeden drobny bug został naprawiony, a wyniki znacząco się poprawiły. Poprzednie pomiary nie zostały ponowione.

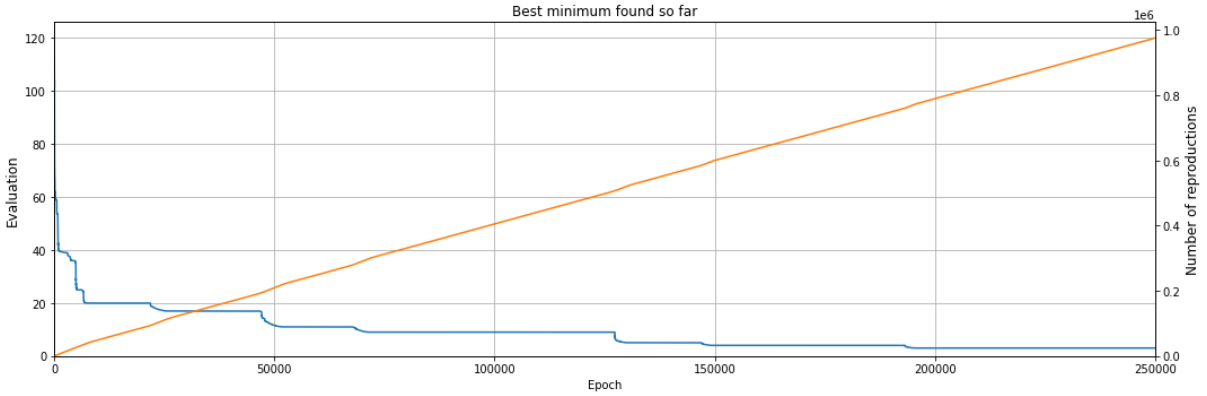
```
@tf.function()
def crossing(a, b, n):
    k = tf.reduce_sum(tf.ones_like(a[:, 0], dtype=tf.int32))

    # Random bubble (hyper-sphere) between points
    reposition = tf.random.uniform([k, n], minval=-1, maxval=1)
    reposition = reposition / tf.norm(reposition, axis=1, keepdims=True)
    new_position_1 = ((a + b) / 2.0) + (tf.maximum((tf.norm(a - b, axis=1, keepdims=True) / 2.0), 1e-3) * reposition)

    # Random exchange of genomes
    new_position_2 = tf.where(
        tf.random.uniform(shape=[k, n]) < tf.random.uniform([k, 1], 0.05, 0.95), a, b)

    # Choose one crossing or another
    new_position = tf.where(
        tf.random.uniform(shape=[k, 1]) < 0.33, new_position_1, new_position_2)

    # Apply mutation, because why not
    # Cauchy distribution derived from uniform distribution
    mutation = 0.05 * tf.math.tan(
        tf.random.uniform(minval=-1.0 * math.pi / 2 + 1e-6, maxval=math.pi / 2 - 1e-6, shape=[k, n]))
    new_position_m = tf.where(
        tf.random.uniform(shape=[k, n]) < tf.random.uniform([k, 1], 0.05, 0.4), new_position + mutation, new_position)

    # Choose mutated or not
    new_position = tf.where(
        tf.random.uniform(shape=[k, 1]) < 0.4, new_position_m, new_position)

    new_position = tf.clip_by_value(new_position, clip_value_min=-5.12, clip_value_max=5.12)

    return new_position
```

Uwagi:

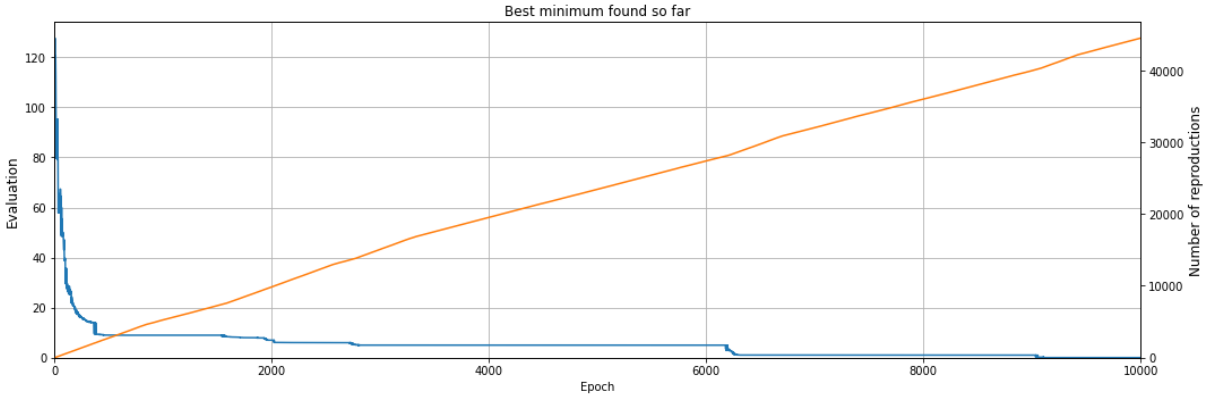
- Mutacje następują z szansą 40%, że wystąpią w ogóle. Jeśli już występują, to występują z prawdopodobieństwem między 5% a 40%.
- Losowy wybór genomu jednego bądź drugiego agenta ma szansę na zaistnienie 67%. Wówczas dla każdego agenta jest losowane prawdopodobieństwo między 5% a 95% tego, że dany element genomu pierwszego agenta będzie przekazany potomkowi.
- Wcześniej był problem z kształtem generowanej macierzy losowej, przez co operacja where trochę nie miała sensu.

Porównanie dla małego problemu (10 wymiarów, 100 agentów)

Zbieżność przed (implementacja z 6 etapu):



Zbieżność po (implementacja z 6 etapu po poprawkach):



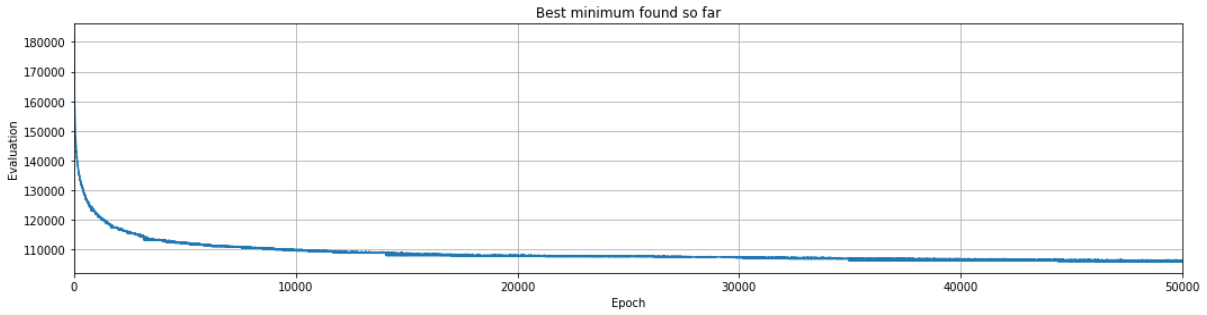
Co istotne teraz udało się zejść praktycznie do rozwiązania prawidłowego (0 na wszystkich wymiarach):

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([ 1.1665196e-04, -2.3897226e-04,  8.6262284e-05, -4.7407349e-05,
       -7.0017690e-05,  1.5582700e-04, -1.3470952e-05, -1.4672516e-04,
        1.7491102e-04, -1.2001489e-05], dtype=float32)>
```

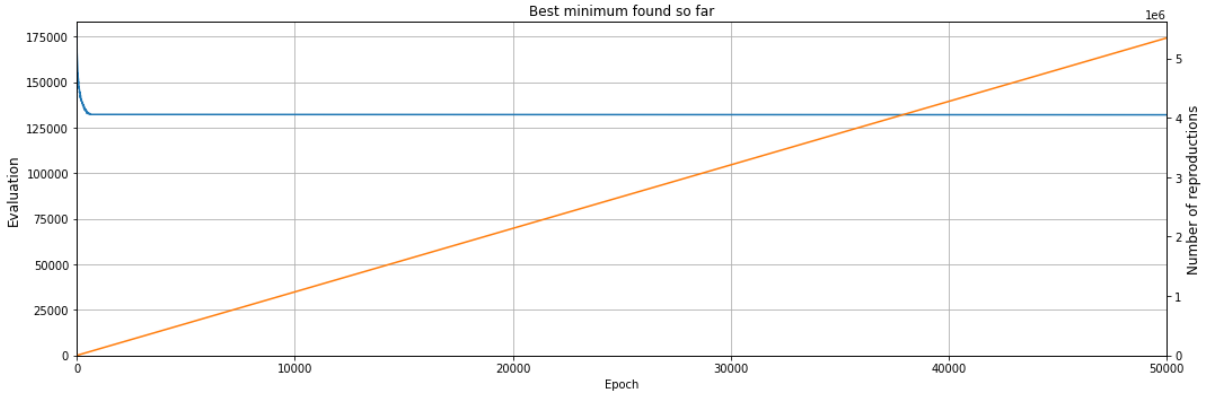
Poprzednio dojście do rozwiązania o ewaluacji ~4 zajęło 250,000 iteracji, podczas gdy teraz prawidłowe rozwiązanie zostało znalezione poniżej 10,000 iteracji.

Porównanie dla dużego problemu (10000 wymiarów, 2000 agentów)

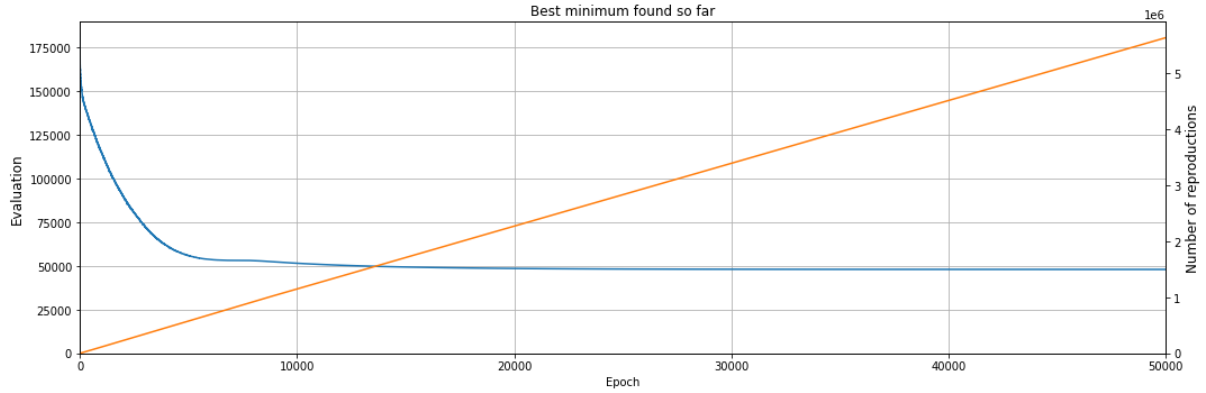
Zbieżność przed (implementacja z 5 etapu):



Zbieżność przed (implementacja z 6 etapu):



Zbieżność po (implementacja z 6 etapu po poprawkach):



Wcześniej dla dużego przykładu (10000 wymiarów, 2000 agentów) po 50,000 epok błąd ewaluacji wynosił 130,000, co było o tyle niepokojące, że był to wynik gorszy niż wersja z etapu 5 (105,000); Teraz błąd spada do około 48,000.

Okazuje się zatem, że faktycznie zastosowanie bardziej zaawansowanych metod krzyżowania dość istotnie pomogło.