Advanced Machine Learning Assignment: 3

Stian Grønlund, Femke van Venrooij (and Monika Kazlauskaitė)

November 2024

Contents

1 Introduction

Note: due to the fact that both of my group partners dropped the course this week, this report is a bit lighter towards the end. I managed to implement all the algorithms, as far as I can tell, but they all seem to run for an incredibly long time, at least in my Python implementations. Therefore, the testing/results is not as complete as i would like it to be.
Combinatoric optimization is a method for finding the optimal solution from a finite space. It includes several algorithms for optimization, like iterative improvement and simulated annealing (SA). Differences in these methods lie in performance and energy efficiency. This report analyzes the performance of the iterative improvement and simulated annealing method, applied to a randomly generated problem modelling an Ising model, and likewise for the specific w500 dataset. The objective is to see for a certain quality solution what the CPU time is and thus the efficiency. The code was implemented in Python.

1.1 Some Mathematics

Part of this week's exercise was also to show some properties of the combinatoric optimization problem for certain versions of the problem. This is worked out in the following two sections.

When x is real

Problem set-up: We have an optimization problem $E = -\frac{1}{2}x^T wx$, with x a free variable, and w a symmetric matrix with 0 on the diagonal.
Because w is a symmetric matrix, we can write $-\frac{1}{2}x^T wx$ as $-\frac{1}{2} \mid\mid Mx \mid\mid^2$, where $m = (M^T M)$, so we have, in essence $-\frac{1}{2}(Mx)^T(Mx)$. We know that the trace of w is zero, and therefore its eigenvalues sum to zero. Therefore, either all eigenvalues are zero, or there is at least one positive and one negative eigenvalue. We can see that what we are really doing is taking a weighted average of the eigenvalues of the matrix w here. If we take a weighted average, then in order to minimize $E(x)$, we should find the highest eigenvalue, λ_{max} we can find, because this will minimize the function. We can find this eigenvalue through Power Iteration, an algorithm which converges to the largest eigenvalue of a matrix (Wong, n.d.). Power iteration has computational complexity $\mathcal{O}(n^2)$ in the dimensions of the input matrix, because we need to do matrix-vector multiplication, which does $n + n$ product and addition operations, n times (for the second dimension of the matrix), resulting in $n(n + n) = 2n^2 \rightarrow \mathcal{O}(n^2)$. We can do this for as many iterations as we want, which would obviously lead to better estimate of the largest eigenvalue, but because that's a scalar term, it's not included in the computational complexity analysis.

When $w_{ij} > 0$ for all i, j

In short, if all values of $w_{ij} > 0$, then, when we take $E = -\frac{1}{2}x^T wx$, we are in essence multiplying each element of the vector x with each other element of the vector, (after adding the positive scalar term from w). In order for this final term to be as large as possible, each multiplication $x_i \times x_j$ should result in a positive number, which is only the case when all x_k are of the same sign. Therefore, E is minimized when all the elements of the vector x are of the same sign.

2 Design

2.1 Iterative Improvement

Iterative improvement implements the technique of local search to find the optimal solution. The pseudo-code for iterative improvement is given in 1.

Algorithm 1 Iterative improvement

start with random initial state x_0
for $t = 1, 2, ...$ **do**
  sample from $x' \in R(x_t)$
  if $E(x') < E(x_t)$ **then**
    accept $x_{t+1} = x'$
  else
    reject $x_{t+1} = x'$
  end if
end for

2.2 Simulated Annealing

Simulated annealing uses the idea of multiple continuing optimization problems to approximate.

$$\min_x E(x) \quad \Rightarrow \quad p_k(x) = \frac{1}{Z} e^{-\frac{E(x)}{T_k}}$$

Algorithm 2 Simulated annealing

Start with a random initial state x_0
for $t = 1, 2, ...$ **do**
  sample state from $x' \in R(x_t)$
  Compute a=exp $\left(-\frac{E(x')-E(x)}{T_t}\right)$
  if $a > 1$ **then**
    accept $x_{t+1} = x'$
  else
    $x_{t+1} = x'$ w.p. a and $x_{t+1} = x_t$ w.p. $1 - a$
  end if
end for

Algorithm 3 Simulated Annealing

Run MH at high temperature to estimate β_1
while $V_k^E > 0$ **do**
  $\beta_{k+1} = f(\beta_k)$ or $\beta_{k+1} = \beta_k + \frac{\Delta\beta}{\sqrt{V_k^E}}$
  Run Metropolis-Hastings (MH) at temperature β_k
  Compute $\langle E \rangle_k$ and V_k^E
end while

3 Implementation

Both implementations made use of Jax's XLA (Bradbury et al., 2018) optimization this week as well. This means that, among other things, any function that relies on randomness will make use of Jax's PRNGkey for pseudorandom generators. Two of the helper-functions needed are also similar between the two algorithms.
First, Listing 1 describes the get_neighbour function, which samples a neighbour within R spin flips of the current state.

Listing 1: get neighbour function

```
1 def get_neighbour(key, state, R):
2
```

```
3       positions = jr.choice(key, len(state), shape=(R,), replace=False)
4       def flip_one(state, pos):
5           return state.at[pos].set(-state[pos])
6       new_state = jax.lax.fori_loop(0, R,
7           lambda i, s: flip_one(s, positions[i]), state)
8
9       return new_state
```

Next, Listing 2 describes the way we create an initial state vector. Sampling from a uniform distribution and doing some arithmetic (lines 4,5) converts the values to −1s and 1s.

Listing 2: create an initial state

```
1  def reset(key, n:int):
2
3      unifs = jr.normal(key, (n,))
4      state = unifs>0.5
5      state = 2*state - 1
6      return state
```

3.0.1 Iterative Improvement

The main iterative improvement algorithm is implemented as detailed in Listing 3. We generate a neighbour (line 8), evaluate the energy of the current and neighbouring states (10, 11), and then decide whether to accept the new state or not (13, 14), using Jax's where function, because Jax does not just-in-time compile Boolean statements that can't be cached.

Listing 3: iterative improvement

```
1  def iterative_improvement(key, data, vec_len, R, num_iterations):
2      state = reset(key, vec_len)
3
4      def body_fun(i, carry):
5          state, key = carry
6          key, subkey = jr.split(key)
7          current_state = state
8          neighbour_state = get_neighbour(subkey, state, R)
9
10         neighbour_energy = energy_calc(neighbour_state, data)
11         current_energy   = energy_calc(current_state, data)
12
13         state = jnp.where(neighbour_energy < current_energy,
14                           neighbour_state, current_state)
15         return (state, key)
16
17     final_state, _ = jax.lax.fori_loop(0, num_iterations-1,
```

```
18                                                  body_fun, (state,key))
19     return final_state
```

This function is further wrapped in a K_num_restarts function, which is in turn wrapped in a N_num_runs function, to do restarts and then average those K restarts over N runs.

3.0.2 Simulated Annealing

The main simulated annealing algorithm is implemented as detailed in Listing 4. We once again use the same Jax-specific where, but notice also the jax.lax.scan function to iterate over the length of the chain.

Listing 4: simulated annealing

```
1  def sim_annealing(key, state:Array, data:Array, R:int,
2                    beta:float, chain_len:int):
3
4      def body_fun(carry, tmp):
5          state, key = carry
6          key, sample_key, prob_key = jr.split(key, 3)
7
8          current_state = state
9          neighbour_state = get_neighbour(sample_key, state, R)
10
11         acc = compute_acceptance(current_state, neighbour_state, data, beta)
12         a = jnp.where(1 < acc, 1, acc)
13         p = jr.uniform(prob_key)
14         state = jnp.where(a > p, neighbour_state, current_state)
15         return (state, key), state
16
17     _, states = jax.lax.scan(body_fun, (state,key), (), length=chain_len)
18     return states
```

This function takes a constant β, and the temperature scheduling is handled in an outer loop. The structure of this outer loop is shown in Listing 5. The structure is the same for both schedules. The structure as shown here is without the extra book-keeping of mean energy, energy variance, and beta values over iterations.

Listing 5: AK schedule

```
1  # first, AK-schedule
2  def ak_body_fn(key, state, beta, data, R, chain_length):
3      states = jit_sim_annealing(key, state, data, R, beta, chain_length)
4      energies = jax.vmap(energy_calc, (0, None))(states, data)
5      mean_E = jnp.mean(energies)
6      var_E  = jnp.var(energies)
7      state = states[-1]
```

```
 8       beta    = AK_schedule(beta, delta_beta, var_E)
 9
10       return state, beta, mean_E, var_E
11
12       delta_beta = 0.1
13
14       beta_0 = 0.0001 # start with high temp=low beta
15       key, init_key, anneal_key = jr.split(key, 3)
16       state = reset(init_key, vec_len)
17
18       current_state, current_beta, mean_E, current_var = ak_body_fn(
19                       anneal_key, state, beta_0, data, R, chain_length)
20
21       for _ in tqdm(generator(current_var)):
22           key, subkey = jr.split(key)
23           current_state, current_beta, mean_E, current_var = ak_body_fn(subkey,
24                           current_state, current_beta, data, R, chain_length)
```

4 Testing and Results

The testing done is slightly different depending on the algorithm ran.
For Iterative Improvement, we first want to check for what K we reach reproducible results. The way I decided to implement this test was in this way: if the standard deviation of the energies of a run was below a certain threshold, then i said the run was finished. Specifically; std_dev < abs(mean)∗eps, where $eps = 0.01$ meaning the standard deviation should be less than 0.1% of the value of the mean. For both the ferromagnetic and anti-ferromagnetic cases, I set up to test values of K from 50 to 5000, in increments of 500. To my surprise, none of these "converged", neither for ferromagnetic, nor for anti-ferromagnetic. The loop for ferro-magnetic ran for 2 hours, and for anti-ferromagnetic. My implementation also meant that I did then not report the value of the energy itself, regrettably.
Next was to estimate running time and energy for different values of K. I set the number of iterations per restart to be 100, K take on one of the values of [20, 100, 200, 500, 1000, 2000, 4000], and running 20 runs and taking the average of them.
The results are in Table 1. As can be seen, while the algorithm I had previously played around with changing the number of iterations as well, (up to several thousand), which did give better returns (lower energy values), though diminishing. For this final run of test results, I did not have time to run at higher number of iterations per restart. As was predicted, the energy is lowest for the highest amount of restarts, and the standard deviation is likewise smaller. The smallest energy value I found for these runs was -2346.0.
For simulated annealing, we were asked to find the number of iterations needed to reach $\mathbb{V}_K E = 0$. I first tried with chain_len= 500 and $\Delta\beta = 0.1$. Unfor-

K	Runtime (s)	Energy	Variance of Energy
20	0.225	-1672.9	105.07
100	0.266	-1838.8	86.94
200	0.227	-1884.8	73.37
500	0.340	-1976.2	84.77
1000	0.219	-1985.0	46.53
2000	0.229	-2053.6	78.87
4000	0.344	-2123.8	65.97

Table 1: Performance metrics across different K values, iterated improvement

$\Delta\beta$	L	CPU (sec)	E
0.1	500	0.03	-6330 ± 90
0.01	500	0.7	-6550 ± 35
0.001	500	7.8	-6570 ± 30
0.001	1000	20.6	-6594 ± 28

Table 2: Table provided by exercise: Performance of SA algorithm on the w500 problem using the AK schedule

tunately, as I am writing this, that is still running, and has reached 140000 iterations in the while loop. I'm not sure it will manage to reach convergence (a global or even local minimum). See the end of the conclusion section for more comments on this.

5 Conclusion

Both of these algorithms seem to take an enormous amount of time to reach an exact solution.
For reference, I will make use of the numbers provided to us by Bert in the slides/exercise text to make a point: Bert provided Table 2 for the Simulated Annealing exercise.
In the slides, it is also mentioned that, for the hyperparameters chosen, it took 15000 iterations to convergence on a minimum. We can do some quick maths to estimate the time it took to compute each of the above energy values by multiplying the number of seconds with iterations. See Table 3 for the values. The total amount of time spent in sequence, if you were to try out all of these combinations, is 118 hours. The total amount of time between assignment deadlines (since the deadline for the previous assignment until the deadline for this one), is 154 hours (7 days). Let's assume that you can do all the sections of the assignment in parallel, and that none of them take more time than this. Let's assume that you lose a further 8hrs on sleep from handing in the previous assignment (at 23:59 Wednesday). That leaves one with 28 hours from Thursday at 07:59 to perfectly implement all code, make sure to run the testing/results code correctly, etc. Then, the code would finish running at 23:59

$\Delta\beta$	L	CPU (sec)	E	Total Time
0.1	500	0.03	-6330 ± 90	7.5 min
0.01	500	0.7	-6550 ± 35	3hrs
0.001	500	7.8	-6570 ± 30	32hrs
0.001	1000	20.6	-6594 ± 28	83 hrs

Table 3: Performance of SA algorithm on the w500 problem using the AK schedule

on Wednesday, leaving 0 hours for writing the report, etc. While I appreciate having assignments and exercises that ask a lot of students, this is a bit much. This is also further assuming that students have access to however much computational resources Bert had.
I have some further, real/actual conclusions. For iterated improvement, I found it to be a surprisingly large factor how many iterations I did per restart. This was surprising because this number was not mentioned in the lecture material at all. Additionally, the time complexity of the algorithm scales as $\mathcal{O}(K \times I \times N \times n^2)$, where I is the number of iterations in the innermost loop, and n^2 comes from the energy calculation. Therefore, there exists a tradeoff here between doing more restarts K, and more inner-loop iterations I.
It seems that, for simulated annealing, using a convergence criterion of strict 0 variance is not an ideal target. While this is the only way to ensure that we have reached an actual minimum, as corroborated by simulated annealing run I currently have going - which is sitting on 100 000 iterations as I write this - there is really no way of knowing when the algorithm will reach zero variance. We could check other things, such as change in mean temperature between runs, or something similar.

6 Appendix

Code for this report can be found at https://github.com/meraxion/cds-ml/tree/main/aml_as-03

References

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/jax-ml/jax

Wong, J. (n.d.). Math 361S Lecture notes Finding eigenvalues: The power method.