TECHNICAL REPORT ON MACHINE LEARNING EXPERIMENTS FOR THE MÖBIUS FUNCTION

DAVID LOWRY-DUDA
LAST UPDATED: 2024.10.19

Abstract. Last week, I was at the Mathematics and Machine Learning program at Harvard's Center of Mathematical Sciences and Applications. The underlying topic was on number theory and I've been studying various number theoretic problems from a machine learning perspective.

This is a technical report, including details related to actually running the code and analyzing the results.

Contents

1. Introduction 1
2. Int2Int 2
3. Representation 6
4. Guessing $\mu(n)$ from n mod p without using CRT 6
5. Extended remark on the utility of ML for pure mathematics 14
Appendix A. Parsing Logs 15
References 24

1. Introduction

I've been computing several experiments related to estimating the Mobius function $\mu(n)$. Previous machine learning experiments on studying $\mu(n)$ have used neural networks or classifiers. Francois Charton made an integer sequence to integer sequence transformer-based translator, [Int2Int] (available at Int2Int), and I thought it would be fun to see if this works any different.

Initially, I sought to get Int2Int to work. I describe aspects of that and how to run it in various ways here.

I'm splitting my description into two parts: a general report [DLD-General] and a technical report. This is the technical report. This includes many details related to actually running and analyzing the code.

This work was supported by the Simons Collaboration in Arithmetic Geometry, Number Theory, and Computation via the Simons Foundation grant 546235.

2. Int2Int

Int2Int can be found at Int2Int. It is possible to run Int2Int using a CPU, but it's much slower and probably not worth trying.

Francois and Edgar Costa (and to a lesser extent, me) have tried ot make Int2Int as self-contained as possible. It's certainly easier now than it was a few weeks ago — it might be possible for the Reader to experiment with https://github.com/f-charton/Int2Int/ using only the README there.

2.1. Training from data files.

By default, Int2Int expects to be able to generate valid inputs and outputs on the fly. We wanted to use and experiment with data that is nontrivial to compute (such as the Möbius function or data associated to elliptic curves). For that, we've added the ability to train from data files.

> It would be fair to say that running Int2Int with default settings is so easy that the hardest part to get up and running is creating the data files. And this is only as hard as the data is to generate and store.

To train from data files, the file needs ot have a particular format. Recall that Int2Int fundamentall reads and outputs sequences of integers. Each integer is encoded as `s ad ... a0`, where `s` is either `+` or `-` and is the sign, and `ad` through `a0` are the digits in a given base (which defaults to 1000). For example, the number 12345 is encoded as `+ 12 345`.

An array of n integers is encoded as `Vn z1 ... zn`, where the `n` in `Vn` is the actual number. For example, the array $(1, 1234, 1234567)$ is encoded as `V3 + 1 + 1 234 + 1 234 567`.

A datafile should have the input given as an array of the appropriate length, followed by a tab character `\t`, followed by the output. As an even more technical note, the output can be specified by a range of values instead of as an integer or integer array; this is useful with $\mu(n)$ since it can only take 3 values. This has to do with the symbol table that Int2Int uses, and the fact that it uses cross-entropy loss to measure performance.

A complete datafile could be the following.

```
V5 + 1 + 2 + 1 + 3 + 4 + 5\t+ 1
V5 + 0 + 2 + 1 + 3 + 1 + 5\t+ 0
```

This data file has the spec `int[5]:int`. If we wanted more than a single `int` as output, we would have to use `Vn` appropriately. There is an additional datatype called `range` (with python-like semantics). In practice, if we know the output is a single constrained integer, there is a minor boost from using `range` instead of `int`.

2.2. Datafile Generation Scripts.

I generated most of my datafiles using a script that closely looked like the following. This is a `sagemath` script. That is, it's mostly python, but it has inbuilt commands `primes` and `moebius` that I take for granted.

```
primes_100 = list(primes(542))  # generate list of 100 primes

def encode_integer(val, base=1000, digit_sep=" "):
    if val == 0:
        return '+ 0'
    sgn = '+' if val >= 0 else '-'
    val = abs(val)
    r = []
    while val > 0:
        r.append(str(val % base))
        val = val//base
    r.append(sgn)
    r.reverse()
    return digit_sep.join(r)

# Each line has an input, a tab, and an output.
def make_line(n):
    return make_input(n) + "\t" + make_output(n) + "\n"

def make_input(n):
    ret = []
    count = len(primes_100)
    ret.append(f"V{2*count}")
    for p in primes_100:
        ret.append(encode_integer(n % p))  # feed in n mod p
        ret.append(encode_integer(p))      # followed by p
    return ' '.join(ret)

def make_output(n):
    return str(moebius(n))
```

What's left is to determine what family of n to input. For performing regression-like tasks, Francois noted that using a log-distribution tends to work best. This is more like a classification task as the output is one of 0, -1, or 1. Thus I generically uniformly sampled integers n up to some large bound like 10^{13} without repetition. To do this, I generate random integers in the range and check to make sure that I don't generate the same one twice.

```
import random

seen = set()
with open("mu_modp_and_p.txt", "w", encoding="utf8") as
↪ outfile:
    while len(seen) < 10**7:
        n = random.randint(2, 10**13)
        if n in seen:
```

```
            continue
        seen.add(n)
        outfile.write(make_line(n))
```

Note that this creates 10^7 lines, each having approximately $200 \cdot 3 \sim 1000$ characters. The resulting file will be approximately 10GB. Adjust the parameters appropriately!

The slow part is computing $\mu(n)$ for random integers. Generating random numbers (including the 10^{-6} chance of hitting a previously seen number) and writing to the file is fast; computing $\mu(n)$ for a random 12 digit number can be slow-ish.

But in practice, the actual slow part is training the resulting ML model. I didn't work to optimize generation of μ at all.

I note that a sieve could generate all the Möbius values up to N at once. Then you could sample from these values in whatever way makes sense. Something along the following lines would work (and would remove the `sagemath` dependency).

```
def primes_up_to(X):
    """
    A basic implementation of Eratosthenes.
    """
    arr = [True] * (X + 1)
    arr[0] = arr[1] = False
    primes = []
    for p in range(X + 1):
        if arr[p]:  # is prime
            primes.append(p)
            for j in range(p*p, X + 1, p):
                arr[j] = False
    return primes


def mobius_up_to(X):
    "Eratosthenes-like"
    arr = [1] * (X + 1)
    arr[0] = 0
    ps = primes_up_to(X)
    for p in ps:
        for j in range(p, X + 1, p):
            arr[j] *= -1
        for j in range(p*p, X + 1, p*p):
            arr[j] = 0
    return arr
```

2.3. **Making testing and training data.** I then make testing and training data.

```python
import os
def shuffle_and_create(fname, ntrain=1900000, ntest=100000):
    "Shuffle and create test and training files"
    if not fname.endswith(".txt"):
        raise ValueError("Incorrect filename assumption.")
    name = fname[:-4]  # remove ".txt"
    print("shuffling...")
    os.system(f"shuf {name}.txt > {name}.shuf.txt")
    print("making training data...")
    os.system(f"head -n {ntrain} {name}.shuf.txt > {name}.txt.train")
    print("making testing data...")
    os.system(f"tail -n {ntest} {name}.shuf.txt > {name}.txt.test")
    print("done!")
```

2.4. **Running Int2Int.** It's now time to actually run the code. It's necessary to have a python with pytorch installed (not surprisingly) and to have Int2Int somewhere. But a generic run would look like

```
python ../Int2Int/train.py
     --num_workers 0
     --dump_path ~/scratch
     --exp_name dld_mu_modp_and_p_sqfree
     --exp_id 1
     --train_data ./mu_modp_and_p_sqfree.txt.train
     --eval_data ./mu_modp_and_p_sqfree.txt.test
     --local_gpu 1
     --epoch_size 250000
     --operation data
     --data_types "int[200]:range(-1,2)"
     --optimizer "adam,lr=0.00025"
```

This was one of the commands I used when using (n mod p, p) for the first 100 primes (giving 200 inputs total) and output just $\mu(n)$ (one int in a prescribed range).

Most of these are straightforward. The `--optimizer` command is deceptively useful, largely because changing the initial learning rate can have large impacts on the overall performance.

When run in this way, it's almost certain that you'll need to manually stop the experiment before it has a complete run. This is because the default number of epochs to train through is very large. In practice, it's a good idea to sometimes look at the outputs or parse the logs and to see how the behavior is going.

2.5. **Log parsing and graph creation.** For log parsing and graph creation, I used code largely written by someone else (maybe Edgar Costa). This is a pile of code, but it's just parsing the pickled logs from a set of experiments. Log writing and parsing always takes piles of not-very-hard code. This is no exception.

The relevant code is in the appendix.

3. Representation

I thought using residues mod several primes was a good strategy. Other experiments have shown[1] that the base in which numbers are expressed can be very important.

Something like base 100 or base 1000 would allow for almost immediate recognition that $\mu(n) = 0$ if $25 \mid n$ or if $4 \mid n$, as these congruence classes are fixed. I'm more interested in what other sorts of mathematical structures the machine can learn.

In this case, I represented each number in base 1000, but almost never needed to use any number larger than 1000 (as the 100th prime is 541). The Chinese remainder theorem shows that this allows representation for every integer up to approximately $10^{219.67}$. This is large enough to be interesting.

```python
# pure python - uses primes_up_to defined above
import math
from functools import reduce

primes_100 = primes_up_to(1000)[:100]
print(primes_100[-1])
# 541

modulus = reduce(lambda x, y: x*y, primes_100, 1)
print(modulus)
# [...enormous...]
print(math.log(modulus)/math.log(10))
# 219.67...
```

If $n < 10^{219.67}$, then n is uniquely determined by its residues mod p for the first 100 primes p.

4. Guessing $\mu(n)$ from n mod p without using CRT

One of the questions that came up was the following **mathematical** (not programmatic) question.

> How would you guess whether n is squarefree or not given n mod p for lots of primes p?

[1] See *Learning the greatest common divisor: explaining transformer predictions* by François Charton. François also extracted Int2Int from the models used in this paper, more or less. Thank you François.

One way would be to perform the Chinese remainder theorem, reconstruct n, and then actually check. There is no known polynomial-time algorithm to check if an integer is squarefree, so this approach is generically slow.

The "default" algorithm would be to note that about 60.79 percent of numbers are squarefree. So guessing `squarefree` all the time would be right just over 60 percent of the time. I want any algorithm that does better.

The Dirichlet series for squarefree numbers that are divisible by a fixed prime q is

$$\frac{1}{q^s}\prod_{\substack{p \\ p\neq q}}\left(1+\frac{1}{p^s}\right) = \frac{1}{q^s}\frac{(1-1/q^s)}{(1-1/q^{2s})}\frac{\zeta(s)}{\zeta(2s)}, \tag{1}$$

and the series for squarefree numbers that aren't divisible by a fixed prime q is the same, but without q^{-s}. Thus the percentage of integers that are squarefree and divisible by q or not divisible by q are, respectively,

$$\frac{1}{q+1}\frac{6}{\pi^2} \quad \text{and} \quad \frac{q}{q+1}\frac{6}{\pi^2}. \tag{2}$$

A simple application of conditional probability shows that

$$P(\text{sqfree}|\text{q-even}) = \frac{P(\text{sqfree and q-even})}{P(\text{q-even})} = \frac{q}{q+1}\frac{6}{\pi^2}$$
$$P(\text{sqfree}|\text{q-odd}) = \frac{P(\text{sqfree and q-odd})}{P(\text{q-odd})} = \frac{q^2}{q^2-1}\frac{6}{\pi^2}.$$

I use the adhoc shorthand q-even to mean divisible by q, and q-odd to mean not divisible by q.

Let's quickly experimentally verify this. We make squarefree numbers with yet another Eratosthenes-type sieve.

```
def squarefree_up_to(X):
    """
    Eratosthenes-like.
    """
    arr = [True] * (X + 1)
    arr[0] = False
    ps = primes_up_to(int(X**.5) + 1)
    for p in ps:
        for j in range(p*p, X + 1, p*p):
            arr[j] = False
    ret = []
    for i in range(X + 1):
        if arr[i]:
            ret.append(i)
    return ret


sfree = squarefree_up_to(10_000_000)
```

```
print(len(sfree)/10_000_000)
# 0.6079291

import math
print(6./math.pi**2)
# 0.6079271018540267
```

As an aside, I note that this converges very quickly. Look at how close that is! One useless application of the Riemann Hypothesis is that is would guarantee how quickly the density of the number of squarefree numbers up to X would converge to $6/\pi^2$.

```
def ratio_sqfree_with(filterfunc):
    return sum(1 for n in sfree if filterfunc(n))/len(sfree)

def is_even(x):
    return 1 if x % 2 == 0 else 0
def is_odd(x):
    return 1 if x % 2 == 1 else 0

# even and sqfree
ratio_sqfree_with(is_even)
# 0.3333309756022536

# odd and sqfree
ratio_sqfree_with(is_odd)
# 0.6666690243977463

def is_3even(x):
    return 1 if x % 3 == 0 else 0
def is_3odd(x):
    return not is_3even(x)

ratio_sqfree_with(is_3even)
# 0.2499983961945624

ratio_sqfree_with(is_3odd)
# 0.7500016038054438
```

This agrees with the claim above that $1/(q+1)$ of squarefree numbers are divisible by the prime q, and $q/(q+1)$ are not. The converse probabilities follow from basic probability, but to make sure:

```
sqfree_set = set(sfree)  # for quick inclusion checking

def prob_sqfree_given(filterfunc):
    sqfree_count = 0
    total_count = 0
```

```
    for n in (x for x in range(10_000_000) if filterfunc(x)):
        total_count += 1
        if n in sqfree_set:
            sqfree_count += 1
    if total_count == 0:
        return 0.0
    return sqfree_count / total_count

# P(sqfree | divis by 2)
prob_sqfree_given(is_even)
# 0.4052832

# P(sqfree | not divis by 2)
prob_sqfree_given(is_odd)
# 0.810575

# P(sqfree | divis by 3)
prob_sqfree_given(is_3even)
# 0.45594380881123825
3/4 * 6/math.pi**2
# 0.45594532639052

# P(sqfree | not divis by 3)
prob_sqfree_given(is_3odd)
# 0.6839217683921769
9/8 * 6/math.pi**2
# 0.68391798958578
```

These are very close to the theoretical computations above — again, it turns out that convergence is very quick.

4.1. **Compound Probabilities.** We'll compute joint probabilities theoretically in a moment. But we'll also experimentally find them.

Let's look at the probability using the small-prime strategy for the primes 2, 3, 5: if n is divisible by one of these, guess that n is not squarefree; otherwise guess that n is squarefree.

```
def not_divis_by_small_prime(n):
    for p in (2, 3, 5):
        if n % p == 0:
            return False
    return True

A = prob_sqfree_given(not_divis_by_small_prime)
print(A)
# 0.9498902374725594

```

```
def prob_notsqfree_given(filterfunc):
    notsqfree_count = 0
    total_count = 0
    for n in (x for x in range(10_000_000) if filterfunc(x)):
        total_count += 1
        if n not in sqfree_set:
            notsqfree_count += 1
    if total_count == 0:
        return 0
    return notsqfree_count / total_count


def divis_by_small_prime(n):
    return not not_divis_by_small_prime(n)

B = prob_notsqfree_given(divis_by_small_prime)
print(B)
# 0.5164203621436034
```

The density of numbers not divisible by $2, 3$, or 5 is $(1-1/2)(1-1/3)(1-1/5) \approx 0.2666$. Thus 0.2666 of the time, n isn't divisible by 2 or 3 or 5 and we would guess that n is squarefree; this is correct about 0.9498 of the time. And the 0.7333 of the time when n is divisible by at least one of 2 or 3 or 5, we guess that n is not squarefree; this is correct 0.5164 of the time.

In total, we expect that this strategy is correct with density

$$0.2666 \cdot 0.9498 + 0.7333 \cdot 0.5164 \approx 0.6318.$$

Let's check:

```
not_divis_prob = (1 - 1/2)*(1 - 1/3)*(1 - 1/5)
corr = not_divis_prob * A + (1 - not_divis_prob) * B
print(corr)
# 0.6320123288979917
```

If you look, you'll see that this does better than the naive guess (always guess squarefree) but is worse than guessing based only on mod 2 data. This is because we're ignoring all of the various cross-correlations. Clearly incorporating cross-correlations can never do worse than only using the mod 2 data.

Suppose we look instead at all the probabilities for all 2^ℓ possibilities of n being divisible or not by the first ℓ primes. Here, I use the first 4 primes, and the strategy is simple: compute whether it is more likely for n to be squarefree or not given each divisibility pattern, and guess that one.

```
def binary_to_prime_sets(n, length=4):
    assert length <= 25
    b = bin(n)[2:]
    b = "0" * (length - len(b)) + b
    is_divis = []
```

```
    not_divis = []
    ps = primes_up_to(100)[:length]
    for l, p in zip(b, ps):
        if l == "1":
            is_divis.append(p)
        else:
            not_divis.append(p)
    return is_divis, not_divis

def divis_rules(is_divis, not_divis):
    def filterfunc(n):
        for p in is_divis:
            if n % p != 0:
                return False
        for p in not_divis:
            if n % p == 0:
                return False
        return True
    return filterfunc

def density_given(is_divis, not_divis):
    filterfunc = divis_rules(is_divis, not_divis)
    count = 0
    for n in (x for x in range(10_000_000) if filterfunc(x)):
        count += 1
    return count/10_000_000

correct = 0
exp = 4
for n in range(2**exp):
    is_divis, not_divis = binary_to_prime_sets(n, length=exp)
    ff = divis_rules(is_divis, not_divis)
    psqfree = prob_sqfree_given(ff)
    density = density_given(is_divis, not_divis)
    prob = max(psqfree, 1 - psqfree)
    correct += density * prob
    print(
       is_divis, not_divis, n,
       psqfree, density, prob, density * prob, correct
    )  # my own diagnostics
print(correct)
# 0.7031860000000001
```

Remarkably this almost no better than just 2 alone! Before performing this computation, I had assumed that it would be notably better. Instead, it's close enough that it might actually be the same as using 2 alone, combined with numerical imprecision.

With this set up, we can compute the theoretical probabilities instead of using experimentally determined probabilities.

4.2. **Actual computation.** Let $\{p_1, \dots, p_N\}$ and $\{q_1, \dots, q_D\}$ denote two disjoint sets of primes. We want to compute the density of squarefree numbers that are divisible by each of the p_i and not divisible by any of the q_j. Each of these local conditions are independent; the overall density is the product of the local densities as described in~(1) and~(2). That is, the density of integers divisible by the p_i and not divisible by the q_j is

$$\prod_{p_i} \Big(\frac{1}{p_i + 1} \Big) \prod_{q_j} \Big(\frac{q_j}{q_j + 1} \Big) \frac{6}{\pi^2}.$$

Recall the chain rule from probability, that says

$$P\Big(\bigcap_{i=1}^k E_i\Big) = P\Big(E_1 | \bigcap_{i=2}^k E_i\Big) = P\Big(E_1 | \bigcap_{i=2}^k E_i\Big) P\Big(\bigcap_{i=2}^k E_i\Big),$$

(and which could chain further). I write $P(\text{sqfree}, p_1, p_2, \widehat{q_1}, \widehat{q_2})$ to mean the probability that a number is squarefree, divisible by p_1 and p_2, and not divisible by q_1 or q_2 (with obvious notational generalization). Then

$$P(\text{sqfree} | p_1, \dots, p_N, \widehat{q_1}, \dots, \widehat{q_D}) = \frac{P(\text{sqfree}, p_1, \dots, p_N, \widehat{q_1}, \dots, \widehat{q_D})}{P(p_1, \dots, p_N, \widehat{q_1}, \dots, \widehat{q_D})}.$$

Divisibility by different primes are independent, so this simplifies to

$$P(\text{sqfree} | p_1, \dots, p_N, \widehat{q_1}, \dots, \widehat{q_D}) = \frac{P(\text{sqfree}, p_1, \dots, p_N, \widehat{q_1}, \dots, \widehat{q_D})}{P(p_1) \cdots P(p_N) P(\widehat{q_1}) \cdots P(\widehat{q_D})}.$$

We also have that $P(p) = 1/p$ and $P(\widehat{q}) = (q-1)/q$.

Altogether, we compute that

$$P(\text{sqfree} | p_1, \dots, p_N, \widehat{q_1}, \dots, \widehat{q_D}) = \prod_{p_i} \Big(\frac{p_i}{p_i + 1} \Big) \prod_{q_j} \Big(\frac{q_j^2}{q_j^2 - 1} \Big) \frac{6}{\pi^2}.$$

Note that this generalizes the previous probabilities and is generically straightforward.

Let's quickly check by computing $P(\text{sqfree}|2,3)$ and $P(\text{sqfree}|2,\widehat{3})$:

$$P(\text{sqfree}|2,3) = \frac{1}{2} \frac{6}{\pi^2} \approx 0.3039,$$
$$P(\text{sqfree}|2,\widehat{3}) = \frac{3}{4} \frac{6}{\pi^2} \approx 0.4559.$$

```
divis_by_2_and_3 = divis_rules([2, 3], [])
print(prob_sqfree_given(divis_by_2_and_3))
# 0.30395813920837217

divis_by_2_not_3 = divis_rules([2], [3])
```

```
print(prob_sqfree_given(divis_by_2_not_3))
# 0.45594574559457457
```

Let's now compute the density of the following strategy being correct:

1. Fix a set of primes P.
2. For each partition of P into two disjoint sets of primes $\{p_i\}$ and $\{q_j\}$:
3. Compute $P(\text{sqfree}|p_1,\ldots,p_N,\widehat{q_1},\ldots,\widehat{q_D})$.
4. For integers satisfying this set of prime divisibility rules, guess "squarefree" if this probability is larger than 0.5; otherwise guess "not squarefree".

```
def prob_sqfree_theoretical(ps, qs):
    ret = 6/math.pi**2
    for p in ps:
        ret *= (p / (p + 1))
    for q in qs:
        ret *= (q*q/(q*q - 1))
    return ret

prob_sqfree_theoretical([2], [3])
# 0.45594532639052
```

I assume we use the first ℓ primes and reuse some of the same logic as above.

```
def density_theoretical(ps, qs):
    ret = 1
    for p in ps:
        ret *= (1/p)
    for q in qs:
        ret *= (q-1)/q
    return ret

def strategy(ell):
  correct = 0
  for n in range(2**ell):
      ps, qs = binary_to_prime_sets(n, length=ell)
      psqfree = prob_sqfree_theoretical(ps, qs)
      density = density_theoretical(ps, qs)
      prob = max(psqfree, 1 - psqfree)
      correct += density * prob
  return correct

strategy(1)
# 0.7026423672846756

strategy(10)
# 0.7034137933079656

```

```
strategy(20)
# 0.7034211847385363

strategy(25)
# 0.7034221516869834
```

Computing this for anything much larger would be prohibitively computationally expensive. Without more sophisticated thinking, it seems we've hit a wall. Presumably this continues to grow, but perhaps is strictly bounded.

I pose this as an open question. It's not clear to me how hard it is to answer it.

Question 1. *What is the limiting behavior of this strategy? Can it be shown to be less than* 71 *percent?*

5. Extended remark on the utility of ML for pure mathematics

Machine learning operates as a block box. There may be many problems where it can achieve impressive results, but it might give no clues as to how it actually does its predictions.

But one place where it is useful is acting as a **one-sided oracle** to determine whether the inputs are enough to correctly evaluate an output. For example, if I wanted to determine if a particular set of inputs are sufficient to determine some behavior, I might try to feed these inputs along with known outcomes into a machine learning blackbox. If the ML soup acts with high accuracy using only those inputs, it seems more likely that those variables are indeed significant.

It's "one-sided" because the model might simply fail to model the function well. Failing to obtain high accuracy could reflect merely that the model wasn't strong enough, or there wasn't enough data, or any of a variety of points of failure that are independent of the underlying mathematical question.

And it's an "oracle" because there are no explanations for the insight. We can not ask for the workings behind the curtain.

With regard to the question above, I think I've tried such a variety of models and structures and learning rates that I suspect that knowing $n \bmod p$ for the first 100 primes isn't enough to guess $\mu(n)^2$ on random input n more than 75 percent of the time. And this belief is bolstered by the failure of the ML to do better. (I've tried neural networks of various forms too, even though I haven't described those here).

But unfortunately that's not the direction the oracle sees and I don't believe in the strength of ML enough to make a conjecture or to draw a line in the sand.

Appendix A. Parsing Logs

```
# path and env name : THIS IS YOUR DUMP PATH
path = "~/scratch/"

# THE EXPERIMENTS YOU WANT TO PROBE AND THE ACCURACY INDICATOR
indicator = "valid_arithmetic"
xp_env=["dld_mu_modp_and_p_sqfree"]

# SET TO TRUE IF YOU USE BEAM SEARCH
has_beam=False

import os
import pickle
import matplotlib.pyplot as plt
import glob
import ast
from datetime import datetime
from tabulate import tabulate
import numpy as np
from operator import itemgetter

xp_id_filter=[]
xp_id_selector=[]
unwanted_args = ['dump_path']
var_args = set()
all_args = {}

# list experiments
xps = [(env, xp) for env in xp_env
       for xp in os.listdir(path+'/'+env)
       if (len(xp_id_selector)==0 or xp in xp_id_selector)
       and (len(xp_id_filter)==0 or not xp in xp_id_filter)]
names = [path + env + '/' + xp for (env, xp) in xps]
print(len(names),"experiments found")

# read all args
pickled_xp = 0
for name in names:
    pa = name+'/params.pkl'
    if not os.path.exists(pa):
        print("Unpickled experiment: ", name)
        continue
    pk = pickle.load(open(pa,'rb'))
    all_args.update(pk.__dict__)
```

```
    pickled_xp += 1
print(pickled_xp, "pickled experiments found")
print()

# find variable args
for name in names:
    pa = name+'/params.pkl'
    if not os.path.exists(pa):
        continue
    pk = pickle.load(open(pa,'rb'))
    for key,value in all_args.items():
        if key in pk.__dict__ and value == pk.__dict__[key]:
            continue
        if key not in unwanted_args:
            var_args.add(key)

print("common args")
for key in all_args:
    if key not in unwanted_args and key not in var_args:
        print(key,"=", all_args[key])
print()
print(len(var_args)," variables params out of", len(all_args))
print(var_args)

def vars_from_env_xp(env, xp):
    res = {}
    pa = path+env+'/'+xp+'/params.pkl'
    if not os.path.exists(pa):
        print("pickle", pa, "not found")
        return res
    pk = pickle.load(open(pa,'rb'))
    for key in var_args:
        if key in pk.__dict__:
            res[key] = pk.__dict__[key]
        else:
            res[key] = None
    return res

def get_start_time(line):
    parsed_line = line.split(" ")
    dt = datetime.strptime(parsed_line[2]+' '+parsed_line[3],"%m/%d/%y %H:%M:%S")
    try:
        idx = parsed_line.index("epoch")
        curr_epoch = int(parsed_line[idx+1])
```

```
    except ValueError:
        curr_epoch = ""
    return dt, curr_epoch

def read_xp(env, xp, indics, max_epoch=None):
    res = {"env":env, "xp": xp, "stderr":False, "log":False, "error":False}
    stderr_file = os.path.join(os.path.expanduser("~"), 'workdir/'+env+'/*/'+xp+'.stderr')
    nb_stderr =len(glob.glob(stderr_file))
    if nb_stderr > 1:
        print("duplicate stderr", env, xp)
        return res
    for name in glob.glob(stderr_file):
        with open(name, 'rt') as f:
            res["stderr"]=True
            errlines = []
            cuda = False
            terminated = False
            forced = False
            for line in f:
                if line.find("RuntimeError:") >= 0:
                    errlines.append(line)
                if line.find("CUDA out of memory") >= 0:
                    cuda = True
                if line.find("Exited with exit code 1") >=0:
                    terminated = True

                if line.find("Force Terminated") >=0:
                    forced = True
            res["forced"] = forced

            res["terminated"] = terminated
            if len(errlines) > 0:
                res["error"] = True
                res["runtime_errors"] = errlines
                res["oom"] = cuda
                if not cuda:
                    print(stderr_file,"runtime error no oom")

    pa = path+env+'/'+xp+'/train.log'
    if not os.path.exists(pa):
        return res
    res["log"] = True
    with open(pa, 'rt') as f:
```

```
        series = []
        train_loss=[]
        for ind in indics:
            series.append([])
        best_val = -1.0
        best_xel = 999999999.0
        best_epoch = -1
        epoch = -1
        val = -1
        ended = False
        nanfound = False
        res["curr_epoch"]=-1
        res["train_time"]=0
        res["eval_time"]=0
        res["pred_nr"]=[]
        nb_sig10 = 0
        nb_sig15 = 0
        counter = 0
        counting = False
        for line in f:
            try:
                if counting:
                    counter += 1
                if line.find("Signal handler called with signal 10") >= 0:
                    nb_sig10 += 1
                if line.find("Signal handler called with signal 15") >= 0:
                    nb_sig15 += 1
                if line.find("Stopping criterion has been below its best value for more than") >=0:
                    ended = True
                elif line.find("============ Starting epoch") >=0:
                    dt, curr_epoch = get_start_time(line)
                    if curr_epoch == max_epoch: break
                    res["start_time"] = dt
                    if curr_epoch >0 and curr_epoch == res["curr_epoch"]+1:
                        res["eval_time"] += (dt - res["end_time"]).total_seconds()
                    res["curr_epoch"] = curr_epoch
                elif line.find("============ End of epoch") >=0:
```

```
                dt, curr_epoch = get_start_time(line)
                if curr_epoch != res["curr_epoch"]:
                    print("epoch mismatch",
                     ↪  curr_epoch,"in", env,",", xp)
                else:
                    res["end_time"] = dt
                    res["train_time"] +=
                     ↪  (dt-res["start_time"]).total_seconds()
            elif line.find("- model LR:") >=0:
                loss = line.split(" ")[-5].strip()
                train_loss.append(None if loss == 'nan'
                 ↪  else float(loss))
            elif line.find("- LR:") >=0:
                loss = line.split(" ")[-4].strip()
                if loss == "predictions.":
                    print(line)
                else:
                    train_loss.append(None if loss == 'nan'
                     ↪  else float(loss))
            elif line.find('- test predicted pairs') >=0:
                counter = 0
                counting = True
            else:
                pos = line.find('__log__:')
                if pos >=0:
                    counting = False
                    res['pred_nr'].append(counter/100.0)
                    if line[pos+8:].find(': NaN,') >= 0:
                        nanfound = True
                        line = line.replace(': NaN,',':
                         ↪  -1.0,')
                    dic = ast.literal_eval(line[pos+8:])
                    epoch = dic["epoch"]
                    if not indicator+"_"+indics[0] in dic:
                        continue
                    if not indicator+"_"+indics[1] in dic:
                        continue
                    val = dic[indicator+"_"+indics[0]]
                    xel = dic[indicator+"_"+indics[1]]
                    if xel < best_xel:
                        best_xel= xel
                    if val > best_val:
                        best_val = val
                        best_epoch = epoch
```

```
                        res["best_dic"] = dic
                    for i, indic in enumerate(indics):
                        if indicator+"_"+indic in dic:
                            series[i].append(dic[indicator+"_"+indic])

            except Exception as e:
                print(e, "exception in", env, xp)
                continue
            except:
                print(line)
                continue
        res["nans"] = nanfound
        res["ended"] = (ended or (nb_sig15 > nb_sig10))
        res["last_epoch"] = epoch
        res["last_acc"] = "{:.2f}".format(val)
        res["best_epoch"] = best_epoch
        res["best_acc"] = float("{:.2f}".format(best_val))
        res["best_xeloss"] = "{:.2f}".format(best_xel)
        res["train_loss"]=train_loss
        res["avg_d"] = np.median(res['pred_nr'])
        res["last_d"] = res['pred_nr'][-1] if len(res['pred_nr']) > 0 else -1
        if epoch >=0:
            res["train_time"] /= (epoch+1)
            res["eval_time"] /= (epoch+1)
        res["train_time"] = int(res["train_time"]+0.5)
        res["eval_time"] = int(res["eval_time"]+0.5)

        for i,indic in enumerate(indics):
            res["last_"+indic] = "{:.2f}".format(series[i][-1]) if len(series[i])>0 else '0'
            res["best_"+indic] = "{:.2f}".format(max(series[i])) if len(series[i])>0 else '0'
            res[indic] = series[i]
            if len(series[i])!= epoch + 1:
                print("mismatch in nr of epochs",env, xp, epoch+1, len(series[i]), indic)
    return res

data = []
indics = ["beam_acc" if has_beam is True else "acc","xe_loss"]
```

```
indics.extend(["correct", "perfect", "beam_acc_d1", "beam_acc_d2",
"beam_acc_nb", "additional_1","additional_2","additional_3"])

for (env, xp) in xps:
    res = read_xp(env, xp, indics, None)  # USE THE LAST PARAMETER IF YOU WANT TO LIMIT READ TO N EPOCHS
    res.update(vars_from_env_xp(env, xp))
    data.append(res)

print(len(data), "experiments read")
print(len([d for d in data if d["stderr"] is False]),"stderr not found")
print(len([d for d in data if d["error"] is True]),"runtime errors")
print(len([d for d in data if "oom" in d and d["oom"] is True]),"oom errors")
print(len([d for d in data if "terminated" in d and d["terminated"] is True]),"exit code 1")
print(len([d for d in data if "forced" in d and d["forced"] is True]),"Force Terminated")
print(len([d for d in data if "last_epoch" in d and d["last_epoch"] >= 0]),"started XP")
print(len([d for d in data if "ended" in d and d["ended"] is True]),"ended XP")
print(len([d for d in data if "best_acc" in d and float(d["best_acc"]) > 0.0]),"began predicting")
```

And to make some graphs displaying various things, I would run the following. Or rather, I would run the above and below in a notebook, so the graphs display inline. (Otherwise I guess I would save them).

In practice, it was sufficient to look at the `tail` of the running log and to extract learning rate failures and accuracies on test sets.

```
import numpy as np

def compose(f,g):
    return lambda x : f(g(x))

def print_table(data, args, sort=False):
    res = []
    for d in data:
        line = [d[v] if v in d else None for v in args]
        res.append(line)
    if sort:
```

```
        res = sorted(res, key=compose(float,itemgetter(0)), reverse=True)
    print(tabulate(res,headers=args,tablefmt="pretty"))

def speed_table(data, args, indic, sort=False, percent=95):
    res = []
    for d in data:
        if indic in d:
            line = [d[v] if v in d else None for v in args]
            val= 10000
            for i,v in enumerate(d[indic]):
                if v >= percent and i < val:
                    val = i
            line.insert(1,val)
            res.append(line)
    e= args.copy()
    e.insert(1,'first epoch')
    if sort:
        res = sorted(res, key=compose(float,itemgetter(1)), reverse=False)
    print(tabulate(res,headers=e,tablefmt="pretty"))

def training_curve(data, indic, beg=0, end=-1, maxval=None, minval=None, export_to=""):
    print(indic)
    for d in data:
        if indic in d:
            if end == -1:
                plt.plot(d[indic][beg:],linewidth=1)
            else:
                plt.plot(d[indic][beg:end],linewidth=1)
    plt.ylim(minval,maxval)
    plt.rcParams['figure.figsize'] = [10,10]
    if export_to != '':
       # print(export_to)
        plt.savefig(export_to,bbox_inches="tight")
    plt.show()

def filter_xp(xp, filt):
    for f in filt:
        if not f in xp:
            return False
        if not xp[f] in filt[f]:
            return False
```

```
    return True

def xp_stats(data, splits, best_arg, best_value):
    res_dic = {}
    nb = 0
    for d in data:
        if d[best_arg] < best_value: continue
        nb += 1
        for s in splits:
            if not s in d: continue
            lib=s+':'+str(d[s])
            if lib in res_dic:
                res_dic[lib] += 1
            else:
                res_dic[lib]=1
    print()
    print(f"{nb} experiments with accuracy over {best_value}")
    for elem in sorted(res_dic):
        print(elem,' : ',res_dic[elem])
    print()

xp_filter ={}

# CHANGE THESE TO FILTER THE EXPERIMENTS
#xp_filter.update({"n_enc_layers":[4]})
#xp_filter.update({"enc_emb_dim":[512]})

fdata = [d for d in data if filter_xp(d, xp_filter) is True]

oomtab = [d for d in fdata if d["error"] is True]
print(f"CUDA out of memory ({len(oomtab)})")
print_table(oomtab, var_args)

forcetab = [d for d in fdata if 'forced' in d and d["forced"] is True]
print(f"Forced terminations ({len(forcetab)})")
print_table(forcetab, var_args)

unstartedtab = [d for d in fdata if "last_epoch" in d and d["last_epoch"] < 0]
print(f"Not started ({len(unstartedtab)})")
print_table(unstartedtab, var_args)

crypto = False
```

```
runargs = ["best_acc", "best_epoch","best_xeloss",  "ended", "last_epoch",
"last_acc", "last_xe_loss","nans", "error", "train_time", "eval_time"]

#runargs.extend(["best_acc_d1" , "best_acc_d2"])
for v in var_args:
    runargs.append(v)
runningtab = [d for d in fdata if "last_epoch" in d and d["last_epoch"] >= 0]
print(f"Running experiments ({len(runningtab)})")

#splits = ['n_enc_layers','dec_emb_dim','reload_size']
#xp_stats(fdata, splits, 'best_acc',90.0)
print()
print_table(runningtab, runargs, sort=True)

training_curve(fdata, "beam_acc" if has_beam is True else "acc",0, -1, None, export_to = "")
training_curve(fdata, "perfect")
training_curve(fdata, "correct")

training_curve(fdata, "xe_loss", 0) #, None, 0.9* np.min([x for d in fdata for x in d["xe_loss"] if x >0.0]))
training_curve(fdata, "train_loss",0, -1, 2)
speed_table(runningtab, runargs, "beam_acc" if has_beam else "acc", sort=True,percent=99)
speed_table(runningtab, runargs, "beam_acc" if has_beam else "acc", sort=True,percent=50)
speed_table(runningtab, runargs, "beam_acc" if has_beam else "acc", sort=True,percent=55)
speed_table(runningtab, runargs, "beam_acc" if has_beam else "acc", sort=True,percent=60)
```

References

[DLD-General] David Lowry-Duda, *General Report on Machine Learning Experiments for the Möbius Function*. 2024 October 21. (Cited on page 1)

[Int2Int] *Int2Int* Github Repository, https://github.com/f-charton/Int2Int. Accessed 2024 October 20.
(Cited on page 1)