LINKÖPINGS UNIVERSITET

TDDI16: Datastrukturer och algoritmer

Lab 2: Hitta dubbletter

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2023

2023-08-09

1 Bakgrund till problemet

För att träna neurala nätverk behövs (oftast) en stor mängd data. Exempelvis sammanställde Nilsback och Zisserman[1] en stor samling bilder av blommor för att kunna träna ett neuralt nätverk att klassificera blommor. I just detta fallet består datamängden av 8189 jpg-filer i en mapp.

Ett problem med stora datamängder av den här typen är att det är svårt att få en bra översikt över vad datan innehåller. Exempelvis är det svårt att säkerställa att alla bilder är unika, och att det inte finns dubbletter. Det är viktigt att känna till om det finns dubbletter inom maskininlärning eftersom det kan påverka både inlärningsprocessen och verifieringsprocessen. Finns exempelvis samma bild i både träningsdatan och testdatan så kommer det neurala nätverket med mycket stor sannolikhet svara rätt på bilden (den har ju sett ”facit”), vilket kan ge en missvisande bild om hur bra nätverket presterar.

Detta problem är inte bara intressant inom maskininlärning. Det finns många andra situationer där det är intressant att hitta dubbletter. Antag exempelvis att du vill arkivera bilder från Internet (likt the Internet Archive). För att spara på lagringsutrymme är det då intressant att identifiera dubbletter, så att du bara behöver lagra en kopia av varje bild. Ett annat exempel är att söka igenom sin (osorterade) fotosamling efter foton som är nära identiska. I detta fall kanske det inte är intressant att ta bort dubbletter, utan snarare att hitta alla liknande bilder så att du som användare kan välja den bästa.

2 Uppgiften

Målet med laborationen är att implementera ett program som undersöker alla bilder i en given katalog (jpg-filer i detta fall). Programmet ska sedan skriva ut (och visa) alla grupper av bilder som är ”lika”.

Exempelvis finns 7 bildfiler i mappen `tiny/` i de givna filerna. Bilderna `a1.jpg` och `a2.jpg` är nästan identiska. Detsamma gäller för bilderna `b1.jpg`, `b2.jpg`, och `b3.jpg`. Resterande bilder är unika. Programmet ska därmed rapportera just detta: att bilderna `a1.jpg` och `a2.jpg` är ”lika” varandra, och att de tre bilderna `b1.jpg`, `b2.jpg`, och `b3.jpg` är ”lika” varandra.

Vid första anblick verkar detta väldigt enkelt. Dock uppkommer snabbt två viktiga frågor:

1. Vi är egentligen inte intresserade av exakta dubbletter (därav att ”lika” är inom citationstecken). Vi vill betrakta två bilder som ”lika” även om det finns små skillnader (exempelvis komprimeringsartefakter, lite olika ljusstyrka, etc.). Detta gör att vi inte bara kan jämföra bilderna pixel för pixel, utan vi måste hitta ett sätt att bestämma om två bilder är ”tillräckligt lika”.
2. Om vi har ett stort antal bilder (exempelvis de 8189 bilder som nämndes ovan) så kan vi inte jämföra alla par av bilder. Det skulle ta för lång tid. Vi behöver alltså hitta en bättre metod!

En lösning på problem nummer 1 finns implementerad i den givna koden. Det visar sig dock att denna lösning kräver att man jämför alla par av bilder. Den är därför inte lämplig för att även kunna lösa problem nummer 2. Uppgiften är alltså att hitta en lösning som löser båda problemen ovan samtidigt. En idé på hur detta kan göras finns beskrivet i avsnitt 6.

När du har implementerat din lösning ska du också svara på frågorna som finns i filen `readme.txt` innan du redovisar för assistent.

[1] `https://www.robots.ox.ac.uk/~vgg/data/flowers/102/`

3 Givna filer

Till labben hör ett antal givna filer. Nedan finns en kort beskrivning av de filer som är intressanta att studera närmare:

`slow.cpp` Innehåller en långsam implementation av ett program som hittar duplicerade bilder. Ditt program (`fast.cpp`) ska bli snabbare än detta program.

`fast.cpp` Innehåller ett skelett till ett program som hittar duplicerade bilder. Tanken är att detta program ska vara mycket snabbare (ca 10 gånger snabbare) än programmet i `slow.cpp`.

`image.h` Innehåller en klass som representerar en bild. En bild kan som bekant betraktas som en 2-dimensionell array av pixlar. Klassen innehåller också grundläggande operationer för att manipulera bilder. Exempelvis finns en funktion `shrink` som skalar ner en bild till en lägre upplösning. Det går att lösa labben utan att förstå implementationen i `image.cpp`.

`pixel.h` Innehåller en klass som representerar enskilda pixlar i en bild. Pixelns färg är representerad som tre flyttal mellan 0 och 1. En för var och en av grundfärgerna röd, grön, och blå. Klassen innehåller även funktionen `brightness()` som beräknar ljusstyrkan hos en pixel, samt operationer för grundläggande aritmetik.

`load.h` Innehåller hjälpfunktioner för att hantera filer. Här finns funktionen `list_files` som hittar alla filer av en viss typ i en katalog. Här finns också funktionen `load_image` som läser in en bildfil. Det går att lösa labben utan att förstå implementationen i `load.cpp`.

`window.h` Innehåller en klass som representerar ett fönster som används till att visa text och bilder på skärmen. Detta används för att rapportera vilka bilder som var ”lika”, men det kan också användas för att visa bilder och meddelanden för felsökning.

`tiny/` Innehåller 7 bilder för felsökning.

`small/` Innehåller ca 70 bilder för testning.

`medium/` Innehåller ca 260 bilder för testning.

`/courses/TDDI16/lab2/large/` Innehåller alla 8189 bilder från Nilsback och Zisserman. Totalt är dessa ca 400 MB, så de följer inte med i de givna filerna. Arbetar du på skolans system finns de i den delade mappen `/courses/TDDI16/lab2/large/`, så du behöver inte kopiera dem till din hemkatalog om du inte vill. Arbetar du på en egen dator så finns filerna att ladda ner ifrån följande länk: `https://www.ida.liu.se/~TDDI16/common/lab2_large.zip`

4 In- och utdata

Koden kan kompileras med kommandot `make`. Detta kompilerar båda programmen `slow.cpp` och `fast.cpp`. Vill du endast kompilera ett av dem kan du använda `make slow` eller `make fast` respektive. Första gången kan flaggan `-j` användas för att kompileringen ska gå snabbare (då startas flera kompileringsprocesser parallellt där det är möjligt).

Programmen kan sedan köras på vanligt vis (dvs. `./slow` eller `./fast`). Den katalog som ska genomsökas anges som parameter på kommandoraden. Så för att använda mappen `tiny` kan man alltså köra exempelvis: `./slow tiny`. Om man vill söka genom den stora datamängden kan man även specifiera en absolut sökväg, likt `./slow /courses/TDDI16/lab2/large`. På så vis behöver man inte kopiera bilderna till sin hemkatalog.

De båda programmen ska sedan rapportera alla duplicerade bilder genom att anropa `window->report_match()`. Funktionen tar en `std::vector<std::string>` som parameter. Denna `vector` förväntas innehålla sökvägen till två eller flera filer som är ”lika”. Sökvägarna skrivs både ut i terminalen och visas i fönstret som programmet skapar.

När `report_match` anropas så pausas körningen tills du har klickat i fönstret som visar bilderna. Detta för att du ska kunna se resultatet från körningen ordentligt. Om du inte vill att programmet väntar kan

du ange flaggan `--nopause` på kommandoraden. Om du inte ens vill se fönstret kan du i stället ange flaggan `--nowindow`. Exempelvis: `./slow --nowindow small`

5 Testdata

De givna filerna innehåller tre olika datamängder som är lämpliga att använda i olika stadier. Filerna i katalogerna `tiny/`, `small/`, och `medium/` är namngivna för att enklare se vilka bilder som borde vara ”lika” varandra. Filer med namn som börjar med `x` är bilder som ska vara unika. Resterande filer är ”lika” alla andra filer som börjar med samma bokstav. Exempelvis är alltså filerna `a1.jpg` och `a2.jpg` ”lika”. Detsamma gäller filerna `b1.jpg`, `b2.jpg`, och `b3.jpg`.

Den lösning som föreslås nedanför kommer att hitta precis dessa dubbletter om den är korrekt implementerad. Ett korrekt program bör alltså skriva ut följande rader om man kör `./fast tiny`. Kodskelettet skriver ut lite ytterligare ledtext. Ordningen spelar ingen roll.

```
MATCH: tiny/a1.jpg, tiny/a2.jpg
MATCH: tiny/b1.jpg, tiny/b2.jpg, tiny/b3.jpg
```

Notera: i och med att labben använder en heurustik för att hitta bilder som är lika, så kan det vara så att din lösning hittar flera bilder som är lika, eller att den missar någon, beroende på vilka parametrar du använder för `create_summary`-funktionen som beskrivs nedan. Om du tycker att du har hittat en heurustik som fungerar bättre kan detta också vara okej. Prata i så fall med din assistent om detta.

6 Lösningsidé

Som nämndes i avsnitt 2 så finns två större problem som måste lösas för att kunna bygga en effektiv lösning på problemet. Det första är att vi egentligen inte är intresserade av att hitta bilder som är exakt lika, utan vi nöjer oss med bilder som är ungefär lika. Det andra är att vi inte vill behöva jämföra alla par av bilder, eftersom det tar lång tid.

Programmet `slow.cpp` löser det första problemet. För att inte kräva att bilderna är exakt lika skalas först alla bilder ner till 32x32 pixlar (som en bieffekt gör detta att programmet använder mindre minne än om alla bilder behövde sparas i full upplösning i RAM), och ljusstyrkan hos bilderna normaliseras. Sedan undersöks alla par av bilder. För varje par av bilder itererar programmet igenom alla pixlar i bilderna. För varje par av pixlar så beräknas skillnaden i ljusstyrka. Skillnaden kvadreras, och medelvärdet av de kvadrerade värdena beräknas. Om medelvärdet är tillräckligt litet så betraktas bilderna som lika. Processen kan sammanfattas med följande formel (x och y är koordinater, a_{xy} och b_{xy} är ljusstyrkan hos individuella pixlar i de två bilderna):

$$d = \sum_{x=0}^{31} \sum_{y=0}^{31} (a_{xy} - b_{xy})^2$$

Som nämndes i avsnitt 2 så är inte denna lösning lämplig för att lösa problem nummer 2, eftersom den kräver att vi jämför bilderna parvis. Vi behöver alltså en annan idé till programmet `fast.cpp`.

För att inte behöva undersöka alla par av bilder vill vi i `fast.cpp` i stället beräkna en ”sammanfattning” av varje bild (representerat i koden av typen `Image_Summary`). Idén är att denna sammanfattning ska innehålla precis lagom mycket information om bilden, så att om sammanfattningen för två bilder är densamma, så är de ungefär lika. Detta gör att vi kan lagra de beräknade sammanfattningarna i en hashtabell (`std::unordered_map` eller `std::unordered_set`) för att snabbt kunna hitta bilder som är ungefär lika.

En stor fråga som kvarstår är då: hur beräknar vi en sådan sammanfattning av en bild? Sammanfattningen måste innehålla tillräckligt mycket information om bilden så att vi kan skilja på bilder som faktiskt är olika. Men den får samtidigt inte innehålla för mycket detaljer som vi som människor finner oviktiga.

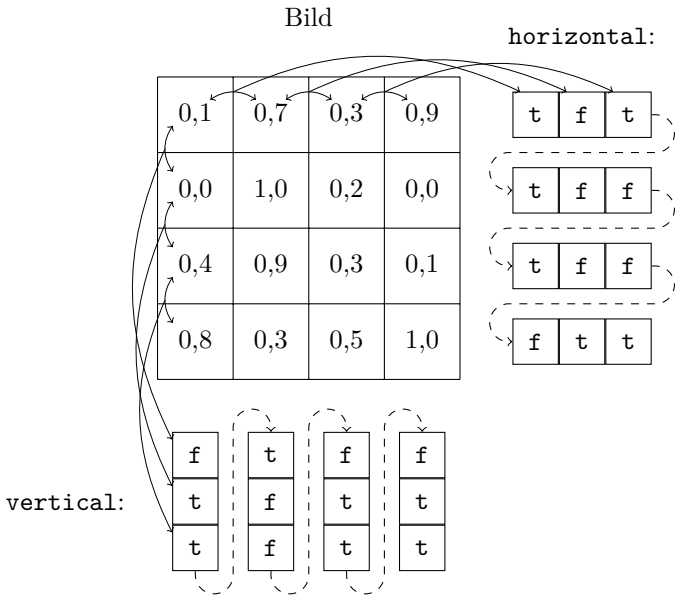
Misslyckas vi med att ”kasta bort” dessa oviktiga detaljer så kommer vi att missa bilder som bara skiljer sig i små, oviktiga detaljer.

Ett sätt att göra detta på är enligt algoritmen nedan. Idén är att först skala ner bilden för att eliminera onödiga detaljer. Sedan undersöker vi förändringar i ljusnivån hos alla rader och kolumner i den nedskalade bilden. Ett sätt att se det på är att vi hittar horisontella och vertikala *konturer* i bilden. Mer konkret kan detta implementeras enligt nedan:

- Välj ett tal, `summary_size`. Valet `summary_size = 8` fungerar bra i detta fall, men testa gärna andra val.
- Skala ner bilden till en bredd och höjd av `summary_size + 1` med hjälp av medlemmen `shrink`. Alltså 9x9 om `summary_size` är 8.
- Iterera genom alla rader av pixlar. För varje par av pixlar, spara en 1:a (`true`) i `horizontal` om den högra är ljusare än den vänstra. Spara annars en 0:a (`false`) i `horizontal`.
- Iterera genom alla kolumner av pixlar. För varje par av pixlar, spara en 1:1 (`true`) i `vertical` om den nedre är ljusare än den övre. Spara annars en 0:a (`false`) i `vertical`.

Processen illustreras med ett exempel i fig. 1. I exemplet är `summary_size = 3`, vilket innebär att bilden först skalas ner till 4x4 pixlar. Innehållet i `horizontal` beräknas sedan genom att iterera genom alla rader i bilden. Första elementet är `true` eftersom ljusstyrkan i pixel (1, 0) är högre än i pixel (0, 0) (dvs. pixlarna representerar att det blir ljusare åt höger). Nästa element är `false` eftersom pixel (2, 0) är mörkare än pixel (1, 0) (dvs. det blir mörkare åt höger). Detta görs för resterande par av pixlar i resterande rader, så att `horizontal` till slut innehåller 12 element.

Innehållet i `vertical` beräknas på liknande sätt, men denna gång itererar vi igenom alla kolumner i stället. Första elmentet är `false` eftersom pixel (1, 0) är mörkare än pixel (0, 0) (dvs. det blir mörkare nedåt). Nästa element är `true` eftersom pixel (2, 0) är ljusare än pixel (1, 0) (dvs. det blir ljusare nedåt). På samma sätt görs detta för resterande par, så att `vertical` innehåller 12 element.



Figur 1: Illustration över hur en nedskalad bild kan ”sammanfattas”. Bilden här är 4x4 pixlar stor. Varje cell innehåller ljusstyrkan av pixeln. I `horizontal` och `vertical` indikeras värdet `true` med `t` och `false` med `f`.

7 Använda `Image_Summary` i `std::unordered_{set,map}`

Behållarna `std::unordered_set` och `std::unordered_map` implementeras i allmänhet med hashtabeller. Typen som används som nyckel i dessa behållare behöver därför ha en likhetsoperator och en hashfunktion. Typen `Image_Summary` har varken en likhetsoperator eller en hashfunktion. Dessa måste alltså implementeras.

Likhetsoperatorn implementeras som vanligt, genom att definiera en `operator ==` på lämpligt ställe i koden. Att definiera en hashfunktion är tyvärr inte lika intuitivt. Man måste specialisera `std::hash` för att göra detta. Alternativt kan man skicka med ett funktionsobjekt som parameter till de hashtabeller man skapar. Att specialisera `std::hash` blir ofta smidigast i slutändan, och det kan göras som följer:

```
namespace std {
    // Definiera en typ som specialiserar std::hash för vår typ:
    template <>
    class hash<Image_Summary> {
    public:
        // Typen ska kunna användas som ett funktionsobjekt.
        // Vi behöver därför överlagra funktionsanropsoperatorn (operator ()).
        size_t operator ()(const Image_Summary &to_hash) const {
            // Beräkna hash här...
            return ?;
        }
    };
}
```

För den som är nyfiken finns processen, och flera alternativ, beskrivna här: `https://en.cppreference.com/w/cpp/utility/hash`

Nu när vi vet hur vi kan definiera en hashfunktion är nästa steg att lista ut vad som behövs för att implementera en bra hashfunktion. En hashfunktion ska som bekant omvandla en instans av nyckeltypen (`Image_Summary` i detta fall) till ett heltal (`size_t` är ett 64-bitars heltal på de flesta moderna system). Resultatet från hashfunktionen kan alltså vara mellan 0 och $2^{64} - 1$.

En hashfunktion måste uppfylla villkoret att om två indata a och b är lika ($a = b$) så måste de producera samma hash ($h(a) = h(b)$). Detta är inte svårt att uppfylla – det räcker att se till att funktionen inte använder globala variabler. En trivial (men väldigt dålig) hashfunktion kan exempelvis alltid returnera 0. Detta uppfyller villkoret ovan, och hashtabellen kommer därmed fungera korrekt.

Anledningen till att vi inte alltid vill returnera 0 är att trots att hashtabellen kommer att fungera så kommer prestandan att vara dålig. I och med att hashfunktionen alltid returnerar 0 så kommer hashtabellen inte att kunna skilja på olika nycklar, och det kommer därför se ut som att alla nycklar kolliderar med varandra. Att hantera kollisioner är dyrt, och de flesta operationer på hashtabellen kommer därmed ta linjär tid i stället för konstant tid.

För att utnyttja styrkan hos en hashtabell måste vi alltså ge den en bra hashfunktion. En bra hashfunktion försöker exponera så mycket information i nyckeln som möjlight i det beräknade hashvärdet. Detta så att om vi har två olika nycklar ($a \neq b$), så ska sannolikheten att de har olika hashvärden vara hög ($h(a) \neq h(b)$). Det gör såklart inget om ett fåtal kollisioner finns, men hashtabellen kommer att prestera bättre om vi kan hålla antalet kollisioner lågt.

En idé till hur man kan tänka när man bygger hashfunktioner kan illustreras med följande sekvens av hashfunktioner. Den första är sämst, och varje steg nedanför blir bättre och bättre:

- `return 0;`
- `return to_hash.horizontal[0];`
- `return 2*to_hash.horizontal[1] + to_hash.horizontal[0];`
- `return 4*to_hash.horizontal[2] + 2*to_hash.horizontal[1] + to_hash.horizontal[0];`
- ...

Mönstret är enklare att se om man tänker binärt. Idén ovan är att vi tänker oss att `to_hash.horizontal` innehåller binära siffror (0 och 1). Vi konverterar dem sedan till ett ”vanligt” heltal genom att byta bas på talet med hjälp av multiplikation och addition (eller bitshift («) och eller (|) om det känns mer logiskt). Om man inser att 64 bitar inte kommer att räcka för utdata är en vanlig teknik att ”blanda” två tal med hjälp av XOR-operatorn (`x ^ y`).

8 Tips

Nedan följer ett par tips för hur man kan bygga lösningen i mindre steg, och testa dessa steg på vägen så att det går att hitta eventuella buggar så tidigt som möjligt:

- Om du använder datamängden `tiny/` så kan du testa din implementation av `compute_summary` genom att för varje bild skriva ut innehållet i `Image_Summary` och manuellt jämföra utskriften för de 7 bilderna.
- För att testa koden som använder en hashtabell (`std::unordered_`...) för att hitta dubbletter utan att designa en bra hashfunktion, kan man tillfälligt använda en hashfunktion som alltid returnerar 0. Detta gör såklart att hashtabellen blir långsam, men i och med att hashtabellerna ger korrekta resultat ändå duger detta för att felsöka koden med mindre datamängder (exempelvis `tiny/` och `small/`).
- Det finns olika varianter av hashtabeller i C++: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset`, och `std::unordered_multimap`. Fundera på vilken av dem som passar bäst för det du försöker göra. Värt att notera kan vara att `std::unordered_multimap` kan upplevas som icke-intuitiv att använda (`[]`-operatorn och medlemmen `at()` finns inte). I stället för `std::unordered_multimap<K, V>` kan det därför vara enklare att använda `std::unordered_map<K, std::vector<V>>`.
- Om din lösning är långsam kan det vara intressant att undersöka om din hashfunktion har många kollisioner. Detta går att göra exempelvis genom att helt enkelt skriva ut resultated av hashfunktionen för alla bilder i `medium/` till standard output, och sedan hitta dubletter enklare genom att köra resultatet igenom kommandot `sort` (exempelvis: `./fast --nowindow medium | sort` om hashvärdet skrivs ut först på raderna). Då ser man enklare hur många anrop till hashfunktionen som ger samma resultat, och kan fundera på varför så är fallet.