# Part C Project Notes

Catherine Vlasov

November 22, 2018

# Contents

# Chapter 1

# Task Documentation

All timings mentioned here are approximate. The specific results can be found in Section 3.2.

## 1.1 Image Curation

### 1.1.1 Initial Image Selection

The first step was selecting which images to use for the experiments. Flickr released a massive database of millions of images and we will use those taken by one user, referred to as `actor00003`. There are $13,349$ images taken by this user and they are on the server under `/array/vlasov/actor00003`. The largest image size in this directory is $3072 \times 2304$ pixels and information about all the images is in a file called `metadata.txt` in the same directory.

I wrote a script called `initial_curation.py` to do the initial image filtering. The script uses the metadata file to identify the images that are $3072 \times 2304$ pixels, makes all of these images grayscale, rotates the portrait ones to landscape, and places the resulting images in a new subdirectory called `size3072`. The script took around 10 minutes to run and 9539 grayscale, $3072 \times 2304$ pixel, landscape images were produced.

### 1.1.2 Choosing Image Sizes

The largest size is $3072 \times 2304$ since that is the largest size we have from `actor00003` and the smallest size was somewhat arbitrarily chosen to be $360 \times 240$. In the graphs with my experiment results, I will want the image sizes (specifically the total number of pixels) to be evenly distributed along the x-axis. In order to achieve this, I picked sizes such that the difference between the number of pixels in consecutive image sizes is roughly the same. I calculated this interval using:

$$\frac{3072 \cdot 2304 - 320 \cdot 240}{9} \approx 777,899 \text{ pixels}$$

It is straightforward to compute the total number of pixels in the $n^{th}$ image size (where $320 \times 240$ is the $1^{st}$ size and $3072 \times 2304$ is the $10^{th}$ size):

$$320 \cdot 240 + (n-1) \cdot 777,899$$

Given the desired number of pixels (call it $P$), we can find dimensions with a 4:3 ratio that produce approximately $P$ pixels. We do so by solving the following equation for $x$ and then computing $4x$ and $3x$ to get the dimensions:

$$P \approx 4x \cdot 3x = 12x^2$$

The results of these computations are:

| Width | Height | Total pixels |
|---|---|---|
| 3072 | 2304 | 7,077,888 |
| 2912 | 2184 | 6,359,808 |
| 2720 | 2040 | 5,548,800 |
| 2528 | 1896 | 4,793,088 |
| 2304 | 1728 | 3,981,312 |
| 2048 | 1536 | 3,145,728 |
| 1792 | 1344 | 2,408,448 |
| 1472 | 1104 | 1,625,088 |
| 1056 | 792 | 836,352 |
| 320 | 240 | 76,800 |

### 1.1.3 Cropping

I wrote a script called `crop.py` to crop the original cover photos to the sizes computed in Section 1.1.2. It takes the desired height and width as well as source and destination directories as commandline arguments. Then it computes where the cropping should start so that $8 \times 8$ blocks are cropped evenly from the top/bottom and left/right and then runs `jpegtran` to do the cropping on all the images in the given source directory. It took around 6 minutes to crop to the smallest size ($320 \times 240$) and 11 and a half minutes to crop to the second-largest size ($2720 \times 2040$).

### 1.1.4 Costs

TO DO

### 1.1.5 Features

TO DO

# Chapter 2

# Meeting Notes

## 2.1 21/11/18

- What I did:
  - Wrote `feature_sanity_check.py`, which adds up all the features for each image in a directory and outputs how many had the same sum (1346, a constant) and how many had zero sum (due to no features being generated).
    - Ran it on all image sizes and found that the number of images for which the features weren't generated when I ran JRM on 10/11/18 increased as the image sizes increased.
  - Reran `compute_features.py` on all images of all sizes.
    - This time no images were skipped, which is good. However, the timing information was lost because for some reason the terminal output was not redirected to a file by `nohup`. This may have been caused by the fact that `nohup` was run by a Bash script for each file size and most likely I forgot the `&` at the end of the line. All I know is that all processes finished after around 18-20 hours. This roughly matches the timing information from the original run on 10/11/18 when some images were skipped.
    - I reran `feature_sanity_check.py` for all sizes. This confirmed that no images were skipped and that the sum of all features in each image of each size was 1346, as expected.
  - Changed `compute_costs.py` so that Python's `multiprocessing` module is used for multithreading in order to spread out the cost computation over multiple processes to speed it up.
    - I initially tried using Python's `threading` module, however creating multiple threads does not actually mean the com-

putation is run in parallel due to Python's global interpreter lock (GIL).
    * `multiprocessing`, on the other hand, avoids the limitations of the GIL and allows multiple processes (as opposed to threads) to be created and make use of multiple cores.
    * Here is a good pro/con analysis of the two modules.
  - Ran `compute_costs.py` on all images of all sizes.
    * All cost files were generated, as expected.
    * The terminal output is messy due to multiple processes printing concurrently. However, the total time is clearly visible on the last line of the terminal output.
  - TL;DR the costs and features of all cover images are all computed!
  - Wrote my own program called JPEG-UTILS, which is based on the J-UNIWARD code and provides a number of checks and information about cover/stego images.
    * Currently it provides two utilities:
      1. Non-zero coefficient count: number of non-zero coefficients in a (cover) image
      2. Differing coefficients: number of coefficients that differ between cover/stego image pairs, broken down by the number that differ by +1, -1, or something else.
  - Looked into how to implement my own binary embedding and it looks like there's no need to reinvent the wheel and the J-UNIWARD code can mostly be reused. Want to discuss.

- *Should I rerun the feature computations just to get the timing information? It takes 18-20 hours.*
  - I can, but it's not a priority now. I could do it later when I'm running the real experiments.

- The `float* pixel_costs` in `cost_model.cpp` of the J-UNIWARD code looks like it's initialized, filled, and then never used, but its "initialization" is actually C pointer arithmetic based on the address of the `float* costs` which is a field inherited from `base_cost_model.cpp`.

- Equations for binary versus ternary embedding

  1. Binary embedding
     - This is the probability of changing coefficient $i$ by $\pm 1$, where $c_i$ is the cost of changing coefficient $i$ (by $\pm 1$) and $c_{\neg i} = 0$ is the cost of not changing it:

$$\pi_i = \frac{1}{1 + e^{\lambda c_i}} = \frac{1}{e^{\lambda c_{\neg i}} + e^{\lambda c_i}} = \frac{e^{-\lambda c_i}}{e^{-\lambda c_{\neg i}} + e^{-\lambda c_i}}$$

- Entropy function for coefficient $i$:

$$-\pi_i \log \pi_i - (1-\pi_i)\log(1-\pi_i)$$

2. Ternary embedding
   - This is the probability of changing coefficient $i$ by $+1$ (for example), where $c_i^0 = 0$ is the cost of not changing coefficient $i$, $c_i^{+1}$ is the cost of changing it by $+1$, and $c_i^{-1}$ is the cost of changing it by $-1$:

$$\pi_i^{+1} = \frac{e^{-\lambda c_i^{+1}}}{1 + e^{-\lambda c_i^{+1}} + e^{-\lambda c_i^{-1}}} = \frac{e^{-\lambda c_i^{+1}}}{e^{-\lambda c_i^{0}} + e^{-\lambda c_i^{+1}} + e^{-\lambda c_i^{-1}}}$$

     * In practice, $c_i^{+1} = c_i^{-1} (= C$, say) for all $i$, meaning that the costs of adding one versus substracting one are the same, so:

$$\pi_i^{+1} = \pi_i^{-1} = \frac{e^{-\lambda C}}{1 + 2 \cdot e^{-\lambda C}}$$

   - Entropy function for coefficient $i$:

$$-\pi_i^{+1} \log \pi_i^{+1} - \pi_i^{-1} \log \pi_i^{-1} - \pi_i^{0} \log \pi_i^{0}$$

- *Can I reuse the structure of the J-UNIWARD code to implement binary embedding?*
  - Yes, but I need to be careful to make sure things are read in the right order and all methods whose implementations are specifically for ternary embedding are adapted to be for binary embedding.
  - Specifically, the entropy function and probability computation need to be changed to use the equations described in the point above.
  - I also need to adapt it so that it takes a number of bits as an argument instead of the number of bits per non-zero coefficient.
- The JRM paper describes how JRM produces 22,510 features. These are actually made up of two sets of 11,255 features, one for the image itself and one for the same image, but with 4-pixel strips cropped off all sides ("Cartesian calibration").
  - The reason the features all add up to the same constant for each image, regardless of size, is that the features are normalized so that they sum to 1. The way JRM computes the features is based on "submodels" and all images have the same number of submodels (672) regardless of size.

  - The features within each submodel (something like this) are normalized, so summing all the features of both halves (from the two versions of the image), makes a total sum of $672 \cdot 1 + 672 \cdot 1 = 1346$, which is what we get when running `feature_sanity_check.py`.

- When I run J-UNWIWARD for the pre-experiments, I should actually use 0.1 instead of 0.4 as previously discussed.
  - For BOSSbase, where all images are $512 \times 512$, $\alpha = 0.2$ produced a "good" classification error rate. However, those images are about $\frac{1}{4}$ of a megapixel whereas the middle size images in our dataset are about 3 megapixels.

- Step-by-step explanation of how we'll choose constants $r_1, ..., r_4$ in order to compute the number of bits of payload to embed. Goal: pick them such that $r_1 = r_2 \cdot \sqrt{n_{mid}} = r_3 \cdot \sqrt{n_{mid}} \cdot \log n_{mid} = r_4 \cdot n_{mid}$, where $n_{mid}$ is the number of pixels in the "middle" image size.
  1. Run J-UNIWARD with $\alpha = 0.1$ on some of the image sizes (among the smaller half of the sizes).
  2. Compute the features of these stego images and then train and run the classifier on these cover-stego image pairs.
  3. Record the classifier's error rate (call it $P_e$) for each image size.
  4. We want to find an image size where $P_e \approx 20\%$. Using `JPEG-UTILS`, see how many non-zero coefficients there were on average and then use this and $\alpha = 0.1$ to compute the average number of bits that were embedded (well, simulated).
  5. Assuming the square root law holds, pro-rate this to compute the number of bits that should be embedded in the middle image size. Call this $m^*_{mid}$.
  6. Knowing $m^*_{mid}$ and $n_{mid}$, compute $r_1, ..., r_4$ so that:

$$\begin{aligned} m^*_{mid} &= r_1 \\ m^*_{mid} &= r_2 \cdot \sqrt{n_{mid}} \\ m^*_{mid} &= r_3 \cdot \sqrt{n_{mid}} \cdot \log n_{mid} \\ m^*_{mid} &= r_4 \cdot n_{mid} \end{aligned}$$

- 95% confidence interval formula (where $P$ is the error rate and $n$ is the sample size):

$$\pm 1.96 \cdot \sqrt{\frac{P \cdot (1 - P)}{n}}$$

- We don't need to worry about splitting the data into training, validation, and testing sets. We can let the classifier handle it using its

default behaviour. We're not too concerned about doing "perfect" machine learning since we can't get many significant figures in our rates in a statistically significant way given that we only have 10,000 images.

## 2.2 14/11/18

- What I did:
  - Ran the hacked J-UNIWARD to compute the costs of all the images of all sizes.
    - It computed the costs of all the $320 \times 240$ images after about 2 hours, but the processes computing the costs for the other sizes either terminated in the middle of their execution (before finishing all images), or did not terminate at all (in the case of $2912 \times 2184$ and $2720 \times 2040$) althought they were not making any progress.
  - Wrote `compute_features.py`, which runs JRM on all the JPEGs in a directory and saves the results to a `.fea` file.
  - Ran `compute_features.py` on all images of all sizes.
    - All ten processes terminated and processed all images.
    - The slowest one took 1h22m and the longest one took 20h4m.
    - However, some of the `.fea` files generated were empty and I don't know why.
  - Was sick the last week, so didn't get much else done.
- The timing of me running J-UNIWARD and JRM (in parallel) coincided with some issues on the server with regard to memory.
  - It ended up being restarted, so I should just rerun everything.
  - Apparently, JRM does sometimes silently fail. However, when I reran it on some of the images whose features were not computed, it worked which was odd.
- It's possible that there was an issue with the simultaneous cost and feature computations because the `compute_features.py` iterated through all files in the directory while more files were being generated both by itself and by J-UNIWARD.
  - When rerunning the computations, I'll modify my script to iterate from 1 to 13349 to "create" the image names itself (and check that they actually exist, since not all 13349 images were used), which will avoid any issues with simultaneous directory iteration and file generation.

- The sums of the features of each image should be a constant, so it would be a good idea to do a sanity check to ensure that this is indeed the case.
- To make the cost and feature computations go faster, it would be a good idea to spin off multiple threads (in Python).
  - For instance, I could spin off 10 threads and make thread 1 handle images with number 1 (mod 10).
- Before running experiments, we need to choose constants $r_1, ..., r_4$ in order to compute the number of bits of payload to embed.
  - Amounts: $r_1, r_2 \cdot \sqrt{n}, r_3 \cdot \sqrt{n} \cdot \log n, r_4 \cdot n$ for each # of pixels $n$.
  - To choose $r_1, ..., r_4$, we do the following:
    1. Run J-UNIWARD with 0.4 bits per non-zero coefficient for all (or every other) image size.
    2. Write and run a script that counts the number of coefficients that differ between the cover and stego images.
    3. From those counts, compute the number of bits that were embedded for each size.
    4. Compute the features of the stego images.
    5. Train the classifier with the features of the cover and stego images.
    6. Run the classifier to determine detectability for each image size.
    7. ...? CLARIFY
  - We'll choose $r_1, ..., r_4$ such that the "middle" image size is in the "middle" among image sizes in terms of detectability. CLARIFY
  - Something about number of non-zero coefficients...

## 2.3 07/11/18

- What I did:
  - Created the file structure on the server.
    - Each `actor00003/sizeXXXX/` directory has one subdirectory called `cover`.
    - We need to decide/calculate how many bits of payload to embed for each size before creating subdirectories for all the different payload sizes.
  - Changed `initial_curation.py` so that constants in the file are instead passed in as command-line arguments.

    - Reran it and it was almost twice as fast (11 vs 19 minutes), though the speedup is probably not related to this change.
  - Finished `compute_probabilities.py`. It finds a $\lambda$ such that:

    $$\sum_{i=1}^{N} H_2(\pi_i) \in [m, m+1), \text{ where } m \text{ is the payload size to simulate}$$

  - Wrote `crop.py` and ran it for the other nine image sizes.
    - It crops 8×8 blocks evenly from the top/bottom and right/left.
    - The process is documented in Section 1.1.3.
  - Refactored the J-UNIWARD code (C++ source files) for readability and uploaded them to my GitHub repository in the `j-uniward` directory.
    - Created a version that integrates Dr. Ker's hack to save the costs to a file (in ASCII) without doing the actual embedding.
    - 99% sure the costs are output in row order, which would be the intuitive way to implement it.
  - Started working on `compute_costs.py`, which runs the hacked J-UNIWARD on all images in a directory.

- Computing the costs for all the images of all sizes
  - On one core, it will take:

    $$50s \times 10000 \times (1 + 0.9 + ... + 0.1) \approx 763 \text{ hours} \approx 32 \text{ days}$$

  - I can use 8-10 of the server's 40 cores.
  - I'll run the script using `nohup` once per directory and obviously the process will finish faster for the smaller images, so I can start on the next task (feature computation) with those that finish first.

- Once the costs are computed, the next task is computing the features using JRM.

- Then, I'll start by using J-UNIWARD with 0.4 bits per non-zero coefficient. I then need to compute the features of the stego objects. Once this is done, I can train the classifier with the features of all the cover and stego objects.
  - The original plan was to use the ensemble classifier.
  - We will instead use a low-complexity linear classifier, which achieves similar performance but has a lower computational complexity.

- When training the classifier, it's important to keep the cover/stego objects pairs together (i.e. not put the cover in the training set and the stego in the testing test for cross-validation).
  - This is important because it ensures the classfier learns the correct boundary.
- The binary embedding I'll write will take the number of bits to embed as input.
  - We will compute the specific number of bits to run the embedding with by manually computing $r \cdot \sqrt{n}$ and $r \cdot \sqrt{n} \cdot \log n$, where $n$ is the total number of pixels in each image and $r$ is some constant we'll pick.
  - We'll pick $r$ such that it makes the detectability of the "middle" image size the median among the detectabilities of all the sizes, where "detectability" refers to the classifier's accuracy.
- When the classifier training function asks for the costs "row-by-row", they mean it should be a big matrix where the first half of the rows are the costs of the cover images, in order, and the second half are the costs of the stego images, in the same order.

## 2.4 31/10/18

- What I did:
  - Put together this document
  - Organized all documents and scripts in my (private) GitHub repository
  - Fixed `initial_curation.py` (the problem is documented in Section 3.3) and I successfully ran it on the server
    - Original images: `/array/vlasov/actor00003/original`
    - All $3072 \times 2304$, grayscale, landscape images are in a new directory `/array/vlasov/actor00003/size3072`
  - Learned how to use `pyplot`, plotted $H_2$ as an exercise
  - Computed the image sizes we'll use (the process and results are described in Section 1.1.2)
    - The method discussed on 24/10/18 doesn't work. It does produce equally sized intervals (in terms of the difference in the total number of pixels between consecutive sizes), but only between $320 \times 240$ and the ninth-largest size since this interval is only around $70,000$. The ninth-largest size would be $960 \times 720$, which is clearly much smaller than $3072 \times 2304$.

    - In order to get sizes linearly distributed in terms of the total number of pixels, the interval needs to be closer to $700,000$ pixels.
  - Started working on `compute_probabilities.py`

- *Is the value of $\lambda$ bounded? How should the binary search (in the context of PLS) work?*
  - $\lambda = 0$ corresponds to maximum entropy (aka. maximum payload) because then $\pi_i = \frac{1}{1+e^{\lambda c_i}} = \frac{1}{2}$
  - As $\lambda \to \infty$, $\pi_i \to 0$
  - The order of magnitude of $\lambda$ depends on the order of magnitude of the costs.
  - The binary search will have two stages:
    1. Exponential search to find an upper bound on $\lambda$. This will involve trying exponentially large values such as $0, 1, 10, 100, ...$ until a value is found such that $\sum_{i=1}^{N} H_2(\pi_i) < M$
    2. Suppose the first value where this inequality holds is $\lambda = 10^n$. We now do a binary search for $\lambda$ with a lower bound of $10^{n-1}$ and an upper bound of $10^n$ and we want to find a value such that $\sum_{i=1}^{N} H_2(\pi_i) \in [m, m+1)$, where $m$ is the number of payload bits.

- In Dr. Ker's paper "On the Relationship Between Embedding Costs and Steganographic Capacity" from June 2018, he writes about how if the detector knows the costs $c_1, c_2, ...c_N$, then the objective that should be minimized is $\sum_{i=1}^{N} c_i \pi_i^2$, which is the same as the objective in PLS except with the $\pi_i$ terms squared.
  - This is a possible project extension.
  - The tricky part is computing the probabilities since the optimal solution is no longer $\pi_i = \frac{1}{1+e^{\lambda c_i}}$. Instead, it's $\frac{\pi_i}{H2'(\pi_i)} = \lambda c_i$.
  - The probabilities can be computed by running Newton-Raphson several times (Dr. Ker did it 8 times)
  - I don't need to tackle this now, but it's worth keeping in mind.

- When I use Dr. Ker's J-UNIWARD hack, I need to make sure that I work out the order in which the costs are written to the file.
  - It's hard to tell just by looking at the costs whether or not they're in the right order. If I'm wrong, I'll probably find out since the embedding will be very detectable.

  - It's very likely that the $8 \times 8$ blocks are analyzed from left to right, top to bottom. However, within each block the costs could be left to right, top to bottom **or** in the zigzag order used to store the quantized coefficients. I need to check this.

- Once I compute the probabilities, it might be a good idea to use Python's `random.seed(..)` method (with the image number as the seed) in order to do the embedding. It can be used to determine whether or not to change each coefficient and so I'll always get the same embedding with the same cover, modulo rounding.

- Dr. Ker has a faster version of JRM for feature extraction.

- Tips:

  - After embedding, open the stego image to make sure nothing got messed up (e.g. due to the order of the costs or coefficients).
  - It would be a good idea to write some scripts to check things like:
    * The number of coefficients that differ between the cover and stego images is $\approx \sum_{i=1}^{N} \pi_i$
    * Coefficients that differ between the cover and stego images only differ by $\pm 1$
  - Test things out on small images (e.g. $64 \times 64$) to save time in case there are bugs.

## 2.5 24/10/18

- What I did:

  - Read Chapter 3 of the Advanced Security notes on steganography
  - Wrote a script (`initial_curation.py`) to find all the largest images in the `actor00003` directory and then make them all grayscale and landscape (described in Section 1.1.1)
    * Wasn't quite working due to "`Empty input file`" error when performing multiple `jpegtran` operations

- Action plan:

  1. Calculate image sizes
     - Preserve the 4:3 aspect ratio, not because we have to but because we can and it means we can keep things as similar as possible
     - The largest image size we'll use is $3072 \times 2304$ since that's the size of the largest `actor00003` images.

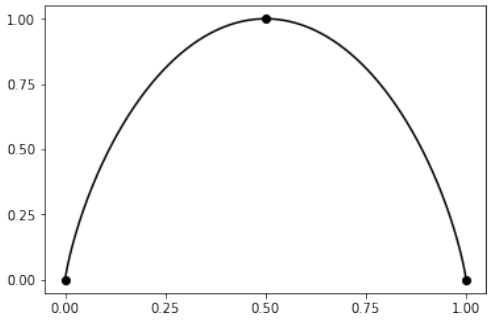
- The smallest size will be $320 \times 240$ since that's a relatively common image size (and it has a 4:3 aspect ratio)
- The short-edge dimensions will be computed by hand by calculating $240x$ (where $x = \sqrt{1}, \sqrt{2}, ..., \sqrt{10}$) and then rounding to the nearest multiple of 24. Then the long-edge dimensions are calculated such that the 4:3 ratio is maintained.

2. Create the directory structure on the server in `/array/vlasov/`
   - Keep a copy of all the original images in `actor00003/original`
   - Create one directory per image size, called `size3072` (for instance)
   - For each size, create two subdirectories:
     (a) One for the unaltered images, called `cover`
     (b) One per number of payload bits, called `stego-1234bits`
   - Each `cover` subdirectory will have three files per cover image:
     (a) `image12345.jpg`: the unaltered image
     (b) `image12345.costs`: the costs computed by J-UNIWARD
     (c) `image12345.fea`: the features computed by JRM
   - Each `stego-1234bits` subdirectory will have one file per stego image:
     (a) `image12345.jpg`: the stego image, which is the cover image `sizeXXXX/cover/image12345.jpg` with a 1234-bit message embedded in it
3. Crop the $3072 \times 2304$ cover images to the sizes calculated in task 1. Do this by cropping $8 \times 8$ pixel blocks evenly from the top/bottom and right/left.
4. Generate the costs (using Dr. Ker's slighty modified J-UNIWARD code) and features (using JRM) for all the cover images of all the different sizes.
   - JRM produces 22510 real numbers (the features)
   - Up to me how to store them, but ASCII is probably the most portable
5. Use J-UNIWARD to embed 0.4 bits per non-zero AC coefficient in some of the covers.
6. Write a function that takes the number of payload bits as input and computes the probabilities with which each coefficient changes during (binary) embedding.
   - Goal: given the costs $c_1, c_2, ...c_N$ (where N is the total number of coefficients) of changing each coefficient (by adding or subtracting one), compute the probabilities $\pi_1, \pi_2, ..., \pi_N$ of making each of these changes

- Size of the payload: $\sum_{i=1}^{N} H_2(\pi_i)$
  - $H_2$ is the "entropy" and is defined as:

$$H_2(x) = -x \cdot \log_2 x - (1-x) \cdot \log_2(1-x)$$

  - Graph of $H_2$:




- Average total cost: $\sum_{i=1}^{N} c_i \pi_i$
- Two (equivalent) optimization problems for computing the payload size:
  (a) Distortion-limited sender (DLS)

$$\text{Maximize } \sum_{i=1}^{N} H_2(\pi_i) \text{ such that } \sum_{i=1}^{N} c_i \pi_i \leq C$$

  (b) Payload-limited sender (PLS)

$$\text{Minimize } \sum_{i=1}^{N} c_i \pi_i \text{ such that } \sum_{i=1}^{N} H_2(\pi_i) \geq M$$

- For some fixed $\lambda$, we can compute the probabilities:

$$\pi_i = \frac{1}{1 + e^{\lambda c_i}}$$

- We'll use PLS, where M is the payload size.
  - The optimal solution is when $\sum_{i=1}^{N} H_2(\pi_i) = M$
  - $\sum_{i=1}^{N} H_2(\pi_i)$ is actually monotonically decreasing, so we can find a value of $\lambda$ such that $\sum_{i=1}^{N} H_2(\pi_i) = M$ for any M we choose. Then, we can compute the probabilities $\pi_1, \pi_2, ..., \pi_N$ using this value of $\lambda$.

    - The end goal is to do the embedding ourselves by modifying each coefficient with these probabilities.

- *Is 80 a standard JPEG quality factor (QF)?* In the massive image database released by Flickr, the most common QFs were 100, the QF used by iPhones, and 80. So, we're using 80 because that gives us a greater selection of images.

## 2.6 17/10/18

- What I did:
  - Read Chapters 1 and 2 of the Advanced Security notes on steganography
  - Read the 2008 paper "The Square Root Law of Steganographic Capacity"
- Discussed questions I had about Chapter 1 (Steganography) and Chapter 2 (Steganalysis) of the Advanced Security notes and about the 2008 paper.
  - *What is downsampling?* Shrinking
  - *When you take a pictures on your phone, what happens?* Captures raw image, immediately compresses it as a JPEG, and discards the raw image
  - *What determines a cover's "source"?* Primarily the camera. The camera's ISO setting, in particular, is very important. The subject of the photos don't make much of a difference.
  - *In JPEG compression, don't you lose some information when dividing the image into* $8 \times 8$ *pixel blocks?* No, the DCT is linear (i.e. 1-to-1 mapping from $8 \times 8$ blocks to coefficients)
  - *Is a JPEG decompressed every time you view it on a computer?* Yes
  - *When LSBR is used on RGB images, which bit(s) are changed?* Good question - it depends, but usually the LSBs of all three components (in sync)
- After embedding a payload, the original cover is destroyed. Otherwise, two nearly identical images would be floating around and Alice could easily be outed if someone got their hands on both versions.

## 2.7 03/10/18

- What I did: N/A
- Discussed software to be used for embedding (J-UNIWARD), feature extraction (JRM), and detection (ensemble of linear classifiers)
  - All the software is here
- Server's IP: 163.1.88.150
- Amounts of payload to embed: $O(1)$, $O(\sqrt{n})$, $O(\sqrt{n}\log n)$, $O(n)$
- $m \sim \frac{\sqrt{DC}}{2}\log\frac{C}{D}$
- TIME EVERYTHING
- I will test new embedding and new detecting methods and I could also try old embedding and new detecting methods
- Total amount of space needed (assuming around 10,000 images are used):
  - Images: $2MB \times 10000 \times 9 \approx 180GB$
  - Costs: $8B \times 5M \times 10000 \approx 400GB$
  - Features: $170KB \times 10000 \times 9 \approx 17GB$

# Chapter 3

# Notes to Self

## 3.1 Useful Commands

- Run a command in the background so that you can keep using the terminal or close it
  - `nohup python script.py &> script_output.out &`
- Check on processes that are running
  - `ps aux | grep vlasov`
- See what processes are currently running, how many resources they're using, etc.
  - `htop`
- View information on how much RAM is used, available, etc.
  - textttfree

## 3.2 Script Timings

- `initial_curation.py` (before I changed constants to command-line arguments)
  - $1131.18478608s \approx 18m51s$ (30/10/18)
- `initial_curation.py --from-dir original/ --to-dir size3072/cover/`
  - $655.185225964s \approx 10m55s$ (01/11/18)
- `crop.py --from-dir size3072/cover --to-dir sizeW/cover/ --width W --height H`, where:

| W | H | Seconds | Approx. Time | Date |
|---|---|---|---|---|
| 2912 | 2184 | 689.414359808 | 11m29s | 01/11/18 |
| 2720 | 2040 | 662.552460909 | 11m02s | 01/11/18 |
| 2528 | 1896 | 662.54279089 | 11m02s | 01/11/18 |
| 2304 | 1728 | 632.872202158 | 10m32s | 01/11/18 |
| 2048 | 1536 | 605.926501989 | 10m05s | 01/11/18 |
| 1792 | 1344 | 555.097690105 | 9m15s | 01/11/18 |
| 1472 | 1104 | 511.460752964 | 8m31s | 01/11/18 |
| 1056 | 792 | 438.49830699 | 7m18s | 01/11/18 |
| 320 | 240 | 359.436480045 | 5m59s | 01/11/18 |

- 10/11/18: `./J-UNIWARD-COSTS -v -I sizeX/cover/ -O sizeX/cover/ -a 0.4`

| Width | Height | Seconds | Approx. Time |
|---|---|---|---|
| 3072 | 2304 | ?? | ?? |
| 2912 | 2184 | ?? | ?? |
| 2720 | 2040 | ?? | ?? |
| 2528 | 1896 | ?? | ?? |
| 2304 | 1728 | ?? | ?? |
| 2048 | 1536 | ?? | ?? |
| 1792 | 1344 | ?? | ?? |
| 1472 | 1104 | ?? | ?? |
| 1056 | 792 | ?? | ?? |
| 320 | 240 | 6430.11 | 1h47m |

  - For some unknown reason, the command for $320 \times 240$ is the only one that finished properly, in the sense that it computed the costs for all $\sim$ 9500 images. The commands for the other nine image sizes either stopped mid-execution after computing the costs for a few hundred images, or continue to run without making any progress (which is the case for $2912 \times 2184$ and $2720 \times 2040$, which have been running for 66 hours but have stopped making progress).

- 10/11/18: `compute_features.py -I sizeX/cover/`

| Width | Height | Seconds | Approx. Time |
|---|---|---|---|
| 3072 | 2304 | 72,282.4331231 | 20h4m |
| 2912 | 2184 | 63,869.6948001 | 17h44m |
| 2720 | 2040 | 54,000.526902 | 15h0m |
| 2528 | 1896 | 40,194.252851 | 11h9m |
| 2304 | 1728 | 32,157.8610289 | 8h55m |
| 2048 | 1536 | 30,295.748122 | 8h24m |
| 1792 | 1344 | 18,653.2866731 | 5h10m |
| 1472 | 1104 | 14,390.2565191 | 3h59m |
| 1056 | 792 | 8,721.86998391 | 2h25m |
| 320 | 240 | 6,430.11 | 1h22m |

  - On 15/11/18, I checked the sums of the features to ensure that they all add up to the same constant. This constant turns out to be 1346 for each image's features, regardless of size.
  - However, it turns out that several hundred images' features were not computed.

- 17/11/18: `compute_costs.py -I sizeX/cover/ -O sizeX/cover/`

| Width | Height | Seconds | Approx. Time |
|---|---|---|---|
| 3072 | 2304 | 202775.362686 | 2d 8h 19m |
| 2912 | 2184 | 194673.499774 | 2d 6h 4m |
| 2720 | 2040 | 183598.260891 | 2d 2h 59m |
| 2528 | 1896 | 172031.831308 | 1d 23h 47m |
| 2304 | 1728 | 152188.728338 | 1d 18h 16m |
| 2048 | 1536 | 129785.996801 | 1d 12h 3m |
| 1792 | 1344 | 104988.846009 | 1d 5h 9m |
| 1472 | 1104 | 74601.1954741 | 10h 43m |
| 1056 | 792 | 39340.753726 | 10h 55m |
| 320 | 240 | 3767.95246005 | 1h 2m |

- 17/11/18: `compute_features.py -I sizeX/cover/`

| Width | Height | Seconds | Approx. Time |
|---|---|---|---|
| 3072 | 2304 | ?? | ?? |
| 2912 | 2184 | ?? | ?? |
| 2720 | 2040 | ?? | ?? |
| 2528 | 1896 | ?? | ?? |
| 2304 | 1728 | ?? | ?? |
| 2048 | 1536 | ?? | ?? |
| 1792 | 1344 | ?? | ?? |
| 1472 | 1104 | ?? | ?? |
| 1056 | 792 | ?? | ?? |
| 320 | 240 | ?? | ?? |

  - Unfortunately the timing information wasn't logged, but as a general idea, all processes finished after around 18-20 hours.

## 3.3 Lessons Learned

- The input and output file to `jpegtran` can't be the same, otherwise you get an "`Empty input file`" error.
- If it looks like directories on the server have disappeared, turn off f.lux or change the colour settings in `.bashrc`.