Two-dimensional classification using SGD
Project report

By Gabriel Samberg

February 2025

1 Abstract

In this project, I attempt to solve the two-dimensional classification problem using the SGD optimization method. This method of optimization is used in order to overcome the computational challenges that inevitably appear when working with large datasets of high-dimensional data. I will first introduce the problem setup, then the mathematical approach to solve it, then dive into the implementation detail, and finally introduce some experimental results.

2 Problem setup

2.1 Prior assumptions

We are given a data set $D = \{X_1, \dots, X_n\} \subset \mathbb{R}^{256\times 256}$ of sampled 2D noisy images with unknown in-plane rotations, translations and scaling

$$X_i = R_{\varphi_i} A + \sigma N_i, \quad i = 1, \dots, n \qquad (1)$$

Where R_{φ_i} denotes the transformations producing the signal part of the i-th image and $N_i \sim \mathcal{N}(0, 1)$ is the noise part of the image. Notice also that the noise level σ is assumed to be unknown and fixed throughout the dataset.

We denote as $\psi, (a, b), c$ the rotation, translation and scale parameters respectively and assume the following prior distributions on them

$$\psi \sim U[0, 2\pi)$$

$$a, b \sim \mathcal{N}(0, (5\%\_of\_image\_size)^2)$$

$$c \sim U[1 - \epsilon, 1 + \epsilon]$$

2.2 Model derivation

Our goal is to approximate the underlying template A in (1).
Motivated by the approach introduced in [1] and [3], instead of just maximizing the log likelihood function

$$\ell(\theta) = log(L(\theta)) = log(P(D|\theta))$$

We assume prior destribution on θ and use Bayes' rule, namely.

$$P(D|\theta) = \frac{P(\theta|D)P(D)}{P(\theta)}$$

To maximize $P(\theta|D)$ with respect to θ. Reminding that we assume that our Data D and our latent variables φ are iid and hence

$$P(D|\theta) = \prod_{i=1}^{n} P(x_i|\theta), \quad p(D,\varphi|\theta) = \prod_{i=1}^{n} P(x_i,\varphi_i|\theta), \quad P(\varphi|D,\theta) = \prod_{i=1}^{n} P(\varphi_i|x_i,\theta)$$

Now, the function we are interested in maximizing is given by

$$\ell(\theta) = log(P(\theta|D)) = log(P(\theta)P(D|\theta)) + C = log(P(\theta)) + log(P(D|\theta)) + C$$

$$= log(P(\theta)) + log(\prod_{i=1}^{n} P(X_i|\theta)) + C$$

$$= log(P(\theta)) + \sum_{i=1}^{n} log(P(X_i|\theta)) + C = log(P(\theta)) + \sum_{i=1}^{n} log(\int_{\varphi} P(X_i|\varphi,\theta)P(\varphi|\theta)d\varphi) + C \quad (2)$$

Where C is the part of the function that is not dependent on θ or σ and hence does not affect the arg-maximum when optimizing with respect to θ and σ.

After experimenting with a couple of different priors on θ I have eventually picked the one also used in [1],

$$P(\theta) = \prod_{i}^{256^2} \lambda e^{-\lambda|\theta_i|}$$

Now, reminding that given θ and φ we have

$$P(X_i|\varphi,\theta) = (\frac{1}{\sqrt{2\pi}\sigma})^J \cdot exp(-\frac{||X_i - R_\varphi A||^2}{2\sigma^2})$$

Since the assumption is that the noise level σ in unknown, we need to optimize both with respect to σ and θ which after getting rid of the elements non dependent on θ and σ brings us to our final loss function

$$\ell(\theta,\sigma) = -\lambda||\theta||_1 - nJ \cdot log(\sigma) + \sum_{1}^{n} log(\int_{\varphi} exp(-\frac{||X_i - R_\varphi\theta||^2}{2\sigma^2})P(\varphi)d\varphi)$$

Where

$$P(\varphi) = P(\psi, a, b, c) = \frac{1}{2\pi} \cdot \frac{1}{2\epsilon} \cdot \frac{1}{2\pi \cdot 0.045^2} exp(-\frac{a^2 + b^2}{2 \cdot 0.045^2})$$

Here note that the variance of the translations was calculated as follows:
for shifting an image by one pixel say in the x direction, we need to pass the parameter $\frac{1}{image\_width}$ to the affine transformation. Hence 5% of the image size is roughly 12 pixelts and hence I took the varicance of the translatioin to be $\frac{12}{256} \approx 0.045$

3 Implementation detail

3.1 Overcoming numerical challenges

Integral evaluation
As it can be seen in the loss function derived in the previous section, we are ought to calculate the integral

$$I = \int_{\varphi} P(X_i|\varphi, \theta) P(\varphi) d\varphi$$

Which has to be numerically approximated. After some experimentation with random sampling based approaches I decided to use the generalized Trapezoidal integration rule with uniform sampling grid [2]. And hence result with

$$\tilde{I} = \sum_{i=1}^{M_\psi} \sum_{j=1}^{M_a} \sum_{k=1}^{M_b} \sum_{m=1}^{M_c} w_{ijkm} P(X_i|\varphi_{ijkm}, \theta) P(\varphi_{ijkm}) \approx I$$

where w_{ijkm} are the integration weights for each sampling point on the grid and $P(\varphi_{ijkm})$ is the multiplication of the densities at this point.

Evaluating the log of the integral
Next, I encountered another numerical challenge when tried to calculate $log(\tilde{I})$ directly. The problem was that in the power of the exponent $P(X_i|\varphi, \theta)$, large numbers appeared that caused $\tilde{I}$ to collapse to zero. One way to solve this issue was to use the numerically stabilized function in PyTorch called log-sum-exp. This required me to re-write the function under the integral sign as

$$-\frac{||X_i - R_\varphi \theta||^2}{2\sigma^2} + log(wP(\varphi))$$

To which I then applied the log-sum-exp function and got a numerically stable evaluation of $log(\tilde{I})$.

Optimizing over σ and evaluating $\log(\sigma)$

Since σ represent the noise level in the model and we are evaluating $log(\sigma)$ in our loss function, we would want it to have strictly positive values. By just establishing a randomly picked and unconstrained model parameter σ, it could attain zero or negative values during the optimization process and hence cause a NaN to appear in the calculation of $log(\sigma)$, breaking the algorithm. In order to avoid this issue, after initializing the model's parameter σ I processed it into a function that mapped it into the positive ray of the real line, this value in turn was passed to the loss function as the new σ. I have experimented with three different functions that were applied to the model's parameter σ. The exponent, the softplus and the sigmoid functions:

$$f(\sigma) = exp(\sigma), \quad g(\sigma) = log(1 + exp(\sigma)), \quad h(\sigma) = \frac{1}{1 + exp(-\sigma)}$$

The function with respect to which the backpropagation and update of σ was the most stable was the sigmoid function. In addition, after processing σ through the sigmoid function I added to it a factor of $1e-12$, so to bound σ away from zero, avoiding some more unstable numerical behavior during optimization when evaluating $log(\sigma)$.

3.2 Validating the algorithm for correctness

Since we had access to the ground truth, validation for correctness was simply achieved by visual comparison between the model's output and the true underlying template for a couple of different examples.

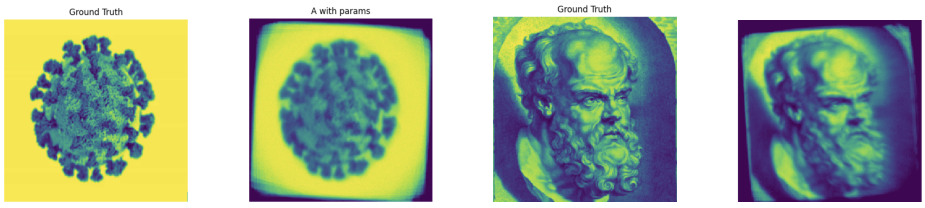


Figure 1: Left:Covid-19 image and it's reconstruction from it's noisy dataset Right: image of Socrates and it's reconstruction from it's noisy dataset

If there was no access to the ground truth, I would check the algorithm for consistency, namely spliting a big data set into two parts and comparing the reconstructions of each part with one another, knowing that the true underlying template is shared between the two smaller datasets. Although here, a sophisticated way of comparing the two reconstructions is needed since the two outputs could differ by a rigid motion and scaling parameter. So, some sort of alignment has to be done and some measure of discrepancy between the two reconstructions has to be calculated.

3.3 The initialization of the algorithm

Big importance with regard to quality of convergence played the way I initialized the parameters of the model. I have experimented with 3 different approaches, the results of which I will show next.

Initialize θ randomly from $\mathcal{N}(0, 1)$
This method didn't yield the best behavior, as can be seen in the figure below.

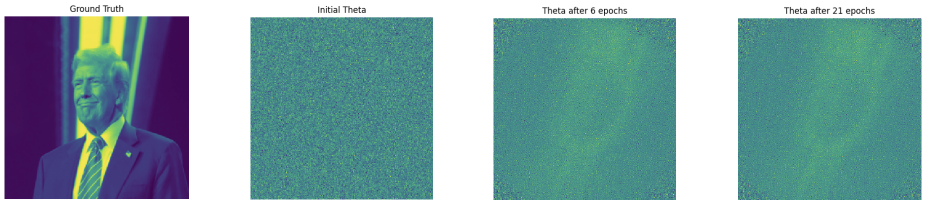


Figure 2: Ground truth on the left and then θ after different number of epochs

Initialize θ with Zero
This method yielded much better results, as can be seen in the figure below.

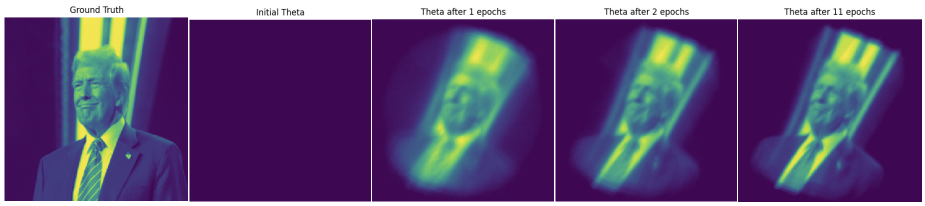


Figure 3: Ground truth on the left and then θ after different number of epochs

Initialize θ with the output $\hat{\theta}$ of optimized lower bound to our loss function

Looking at the term

$$log(\int_{\varphi} P(X_i|\varphi, \theta)P(\varphi|\theta)d\varphi)$$

from (2), we notice that since

$$\int_{\varphi} P(\varphi|\theta)d\varphi = 1$$

And by the convexity of $-log$ and Jensen inequality the following holds:

$$log(\int_{\varphi} P(X_i|\varphi,\theta)P(\varphi|\theta)d\varphi) \geq \int_{\varphi} log(P(X_i|\varphi,\theta))P(\varphi|\theta)d\varphi$$

And hence

$$\sum_{i=1}^{n} log(\int_{\varphi} P(X_i|\varphi,\theta)P(\varphi|\theta)d\varphi) \geq \sum_{i=1}^{n} \int_{\varphi} log(P(X_i|\varphi,\theta))P(\varphi|\theta)d\varphi = Q(\theta)$$

By taking

$$\hat{\theta} = \arg\max_{\theta}(-\lambda||\theta||_1 - \sum_{1}^{n} \int_{\varphi} \frac{||X_i - R_{\varphi}\theta||^2}{2\sigma^2} P(\varphi)d\varphi) = \arg\max_{\theta} Q(\theta)$$

I initialize the algorithm with $\theta_0 = \hat{\theta}$. Notice here that I get $\hat{\theta}$ by optimizing $Q(\theta)$ only over θ and ignore σ by setting $\sigma = 1$. This approach bring to a very good starting point for the optimization of $\ell(\theta, \sigma)$. As can be seen in the figure below.

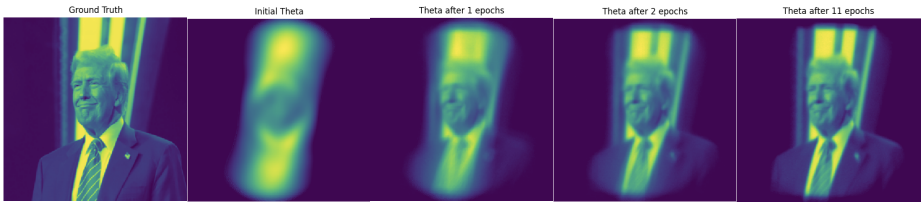


Figure 4: Ground truth on the left and then θ after different number of epochs

3.4 Determining convergence

For determining convergence We can use the sum of relative errors between the current epoch and two previous epochs.
Given the error of epoch i of the optimization process

$$e_i = \ell(\theta_i, \sigma_i)$$

We compute the relative error as

$$relative = |\frac{e_{i-1} - e_i}{e_i}| + |\frac{e_{i-2} - e_i}{e_i}|$$

The stopping criteria can be chosen to be

$$relative \leq \epsilon$$

Note that in practice, I didn't use it since the algorithm converged very fast (just in couple of epochs), and hence almost always was manually stopped or initially was running just for a couple of epochs which was enough.

3.5 Optimization of the algorithm

Adjusting the interval of sampling at every epoch

Computing a 4-dimensional integral at every iteration is computationally heavy, and hence the number of samples for each parameter is very limited (say we are interested in sampling equal number of samples for every parameter, then the number of computations done grows as x^4).

To partially address this issue of heavily limited sampling ability, we want to increase sampling quality.

As can be seen in the figures above, quite after the first iteration, the reconstructed image θ already takes it's position (angle, translation parameter and scale proportion) and is not changed much later on in future iterations. Informally, this means that at this point in the optimization process, many of the parameters on the grid are no longer relevant, and hence we should adjust our sample interval around where its most relevant, namely where $exp(-\frac{||X_i - R_\varphi \theta||}{2\sigma^2})$ is maximal.

Say we are currently looking to adjust the sampling points for the data point X_l and we are holding n sample points for the rotation angles $\{\psi_1, \ldots, \psi_n\}$ equally spaced on the interval $[0, 2\pi]$. Then in order to update the sampling interval, we seek to find the index i s.t

$$\psi_i = \underset{\psi \in \{\psi_1, \ldots, \psi_n\}}{\arg\max} \sum_j^{M_a} \sum_k^{M_k} \sum_m^{M_m} exp(-\frac{||X_l - R_{\psi a_j b_k c_m} \theta||^2}{2\sigma^2})$$

Then, our next sampling points for rotation angles $\{\psi_1, \ldots, \psi_n\}$ will be equally spaced on the interval $[\psi_i - \epsilon, \psi_i + \epsilon]$.

Where ϵ is getting smaller after each epoch but does not vanish.

After experimenting with this approach, I have come to notice that adjusting the intervals for integration had a great effect on convergence when used for the rotation parameters ψ but not so much for the rest of the parameters, and hence in practice I used it only for the rotation sample points.

After experimenting with different values and shrinkage rates of ϵ I set it as follows

$$\epsilon = \frac{20}{360} + (\frac{3}{4})^{epoch\_num}$$

Parallel computation using GPU

Since we are evaluating the expression

$$||X_i - R_\varphi \theta||^2$$

Over a 4-dimensional grid (angles, translation and scale parameters), it would take very long to calculate if doing so sequentially. With the ability of parallel computation on the GPU I have used PyTorch built-in functions Affine_grid(), and grid_sample() that allow to calculate the affine transformations of θ over a grid at once. In addition, the evaluation of the integration weights w_{ijkm} and

the density functions $P(\varphi_{ijkm}) = P(\psi_i)P(a_j)P(b_k)P(c_m)$ over the grid also was implemented in a parallelized manner.

Note here that I have parallelized this computation for each X_i separately. Next optional optimization would be to calculate all the affine transformations over the 4-dimensional grid and adjusting the integration intervals for all the elements in the batch at once, but that requires some structural reorganization of the code and I did not have enough time left unfortunately.

3.6 Setting the hyper-parameters

NOTE:
This section was written last, and unfortunately I had no access to GPU at this point due to some crash that occurred on the cluster I have access to. My own machine is not strong enough to run the code efficiently. Hence, unfortunately limited with time, I have summarized here the process I went through in order to set the hyper parameters of the model but didn't include the actual numerical data like run time comparison and convergence comparisons in numbers.

SGD variant
After experimenting with ADAM, regular SGD, SGD with momentum, I eventually picked momentum-based SGD with Nesterov Accelerated Gradient. This modification of the momentum-based SGD changes the order of computed gradients, first computing the combination of previous computed gradients and making a step in that direction and only then computing the gradient at the current point and making a step in that direction. A nice informal description of the idea of this method is "It is better to correct a mistake after you have made it."
In any case, this method yielded a good result compared to the others with respect to the quality and speed of convergence.

Learning rate
After some experimentation I have noticed that the model is significantly more sensitive to change in σ rather than in θ. In order to address this, I have split the parameters of the models into two groups, θ, and σ. For each group, I assigned different learning rate where the learning rate of σ is significantly lower than the one of θ.

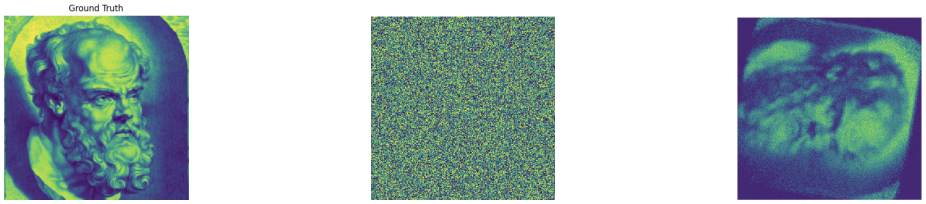
In addition to attaching two different learning rate values to each parameter I have come to notice that gradual decrease in learning rate once every couple of epochs also had a good effect on convergence stability and speed. For that purpose I have used the build-in PyTorch torch.optim.lr_scheduler.StepLR which is a scheduler method that I have set such that to decrease the learning rate every 4 epochs by 90%.
After some experimenting, the initial learning rate for σ was set to 0.001 and

the initial learning rate for θ was set to 0.01.

Batch size
The batch size was chosen to be 5 after experimenting with other batch sizes like 20, 100 and more. The machine on which I was working could not process a batch size of 100 because of the high memory load. I have noticed that with small size batches the algorithm ran allot faster and converged quicker.

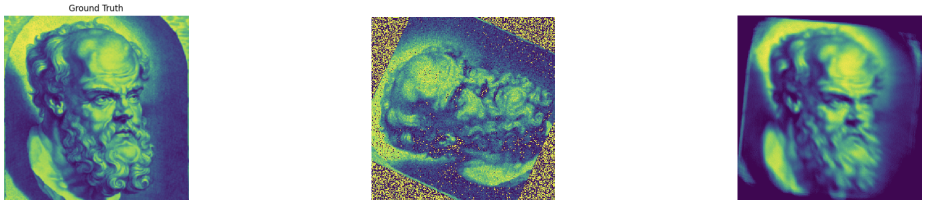
4 experiments

Here I will plot couple of different scenarios for different SNR values and dataset size.



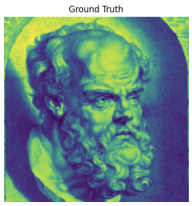
(a) Ground truth (b) Data sample (c) Reco

Figure 5: **SNR = 0.42, Dataset_size = 100**

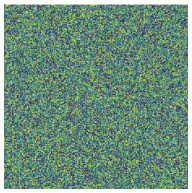


(a) Ground truth (b) Data sample (c) Reco

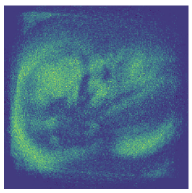
Figure 6: **SNR = 12, Dataset_size = 100**



(a) Ground truth

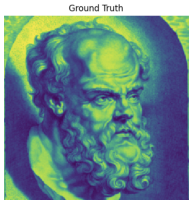


(b) Data sample

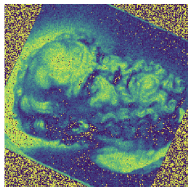


(c) Reco

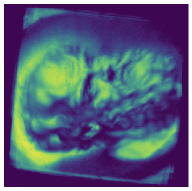
Figure 7: **SNR = 0.42, Dataset_size = 20**



(a) Ground truth



(b) Data sample



(c) Reco

Figure 8: **SNR = 12, Dataset_size = 20**

5 References

[1] A. Punjani, M. A. Brubaker, and D. J. Fleet, "Building Proteins in a Day: Efficient 3D Molecular Structure Estimation with Electron Cryomicroscopy".

[2] D. Keffer, ChE 505 ,University of Tennessee, September, 1999, "Numerical Techniques for the Evaluation of Multi-Dimensional Integral Equations".

[3] Lecture notes on SGD and Maximum Likelihood