Large Scale Machine Learning

European Summer School in Financial Mathematics, Le Mans

S. Gaïffas

1. Stochastic optimization
   - Stochastic Gradient Descent (SGD)
   - Beyond SGD
2. Supervised learning recipes (bis)
   - File indexing
   - Lazy-updates
   - Cross-validation
   - Warm starting
3. Collaborative Filtering
   - Amazon, Google, Netflix
   - Netflix Prize
   - Matrix completion
4. Main Approaches
   - Definitions
   - Measures of success
   - Biases
   - CF as classification / regression
   - K-NN
5. Matrix Factorization
   - Intro
   - Formulation
   - Link with PCA
   - Alternating Least-Squares
   - Gradient Descent
   - Stochastic Gradient Descent
6. A convex formulation
   - Convex relaxation for the rank
   - Proximal gradient descent
7. Illustrations
8. A groundbreaking theory

Quick recap on convex optimization

We want to minimize

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} f_i(\theta)$$

where $f_i(\theta) = \ell(y_i, \langle x_i, \theta \rangle) + \frac{\lambda}{2} \|\theta\|_2^2$

- Ridge Regression
- Ridge Logistic Regression
- etc.

Quick recap on convex optimization

Full (Batch) Gradient Descent

$$\theta^k \leftarrow \theta^{k-1} - \eta_k \nabla f(\theta^{k-1})$$

You've seen:

- If f convex and L-smooth then for $\eta_k = 1/L$

$$f(\theta^k) - f(\theta_*) \leq \frac{2L\|\theta^0 - \theta_*\|}{k+1} \tag{1}$$

  where $\theta_* \in \operatorname{argmin}_\theta f(\theta)$

- Acceleration (Nesterov, Fista): rate improvement $O(1/k^2)$
- Linesearch

If f is also μ-strongly convex, then linear convergence

$$f(\theta^k) - f(\theta_*) \leq \left(1 - \frac{L}{\mu}\right)^k (f(\theta_0) - f(\theta_*)) \tag{2}$$

Stochastic optimization

What if n (and d) is large?

- Each iteration of a full gradient method has complexity $O(nd)$
- I can't put $n \times d$ floats (32 or 64 bits) in my memory

Size of big data makes a modern computer look old: go back to "old" algorithms

- Idea: in machine learning, objective functions are averages of losses

Stochastic Gradient Descent

If I choose uniformly at random $I \in \{1, \ldots, n\}$, then

$$\mathbb{E}[\nabla f_I(\theta)] = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta) = \nabla f(\theta)$$

- $\nabla f_I(\theta)$ is an *unbiased* but very noisy estimate of the full gradient $\nabla f(\theta)$
- Computation of $\nabla f_I(\theta)$ only requires the I-th line of data ($O(d)$ and smaller for sparse data, see next)

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) algorithm (Robbins and Monro 1951)

- At each iteration, load a line of data chosen randomly (requires an index for fast random access on the hard drive)
- Compute gradient for this line of data
- Do a descent step, using this gradient, and repeat

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD)

- **Input**: starting point θ^0, sequence of learning rates $\{\eta_t\}_{t\geq 0}$
- For $t = 1, 2, \ldots$ until *convergence* do
  - Pick at random (uniformly) i in $\{1, \ldots, n\}$
  - Put
$$\theta^t = \theta^{t-1} - \eta_t \nabla f_i(\theta^{t-1})$$
- **Return** last θ^t

- Each iteration has complexity $O(d)$ instead of $O(nd)$ for full gradient methods
- Possible to reduce this to $O(s)$ when features are s-sparse using **lazy-updates** (more on this later)

Stochastic Gradient Descent

- Note that if i is chosen uniformly at random in $\{1, \dots, n\}$

$$\mathbb{E}[\nabla f_i(\theta^{t-1}) | \mathcal{F}_{t-1}] = \frac{1}{n} \sum_{i'=1}^{n} \nabla f_{i'}(\theta^{t-1}) = \nabla f(\theta^{t-1})$$

  where $\mathcal{F}_t =$ information until iteration t (relative to random sampling of indexes)
- Namely, SGD uses very noisy unbiased estimations of the full gradient
- Learning rate η_t usually chosen as $\eta_t \approx Ct^{-\alpha}$ with $\alpha \in [1/2, 1]$.
- Linesearch: η_t is a valid learning rate if

$$f_i(\theta^t - \eta_t \nabla f_i(\theta^t)) \leq f_i(\theta^t) - \frac{\eta_t}{2} \|\nabla f_i(\theta^t)\|_2^2$$

Polyak-Ruppert averaging (ASGD)

- Use SGD iterates $\{\theta^t\}$ but return $\bar{\theta}^t = \frac{1}{t}\sum_{t'=1}^{t}\theta^{t'}$
- Computed "online"

$$\bar{\theta}^t \leftarrow \frac{1}{t}\theta^{t-1} + \frac{t-1}{t}\bar{\theta}^{t-1} \tag{3}$$

- Leads to better results, cf:

`http://leon.bottou.org/projects/sgd`

Stochastic Gradient Descent

Theoretical knowledge on SGD. Typical assumptions

- Each f_i is L-smooth (gradient L-lipshitz)
- f is μ-strongly convex
- Non strongly convex: rate $O(1/\sqrt{t})$ for ASGD with $\eta_t = O(1/\sqrt{t})$
- μ-Strongly convex: rate $O(1/(\mu t))$ for ASGD with $\eta_t = O(1/(\mu t))$

Both SGD and ASGD are **slow**. Best case is $O(1/t)$ convergence when f is strongly convex, while $O(e^{-\rho t})$ for FG

Beyond SGD

Recent results improve this:

- Bottou and LeCun (2005)
- Shalev-Shwartz et al (2007, 2009)
- Nesterov et al. (2008, 2009)
- Bach et al. (2011, 2012, 2014, 2015)
- T. Zhang et al. (2014, 2015)

Stochastic vs Batch optimization

- Gradient descent:

$$\theta^t \leftarrow \theta^{t-1} - \frac{\eta_t}{n} \sum_{i=1}^{n} \nabla f_i(\theta^{t-1})$$

  $O(nd)$ iteration but linear convergence $O(e^{-\rho t})$ (strongly cvx case)

- Stochastic gradient descent:

$$\theta^t \leftarrow \theta^{t-1} - \eta_t \nabla f_{i_t}(\theta^{t-1})$$

  $O(d)$ iteration but slow convergence $O(1/t)$ (strongly cvx case)

Do a fast algorithm with $O(d)$ iteration exist?

Variance reduction

- Put $X = \nabla f_I(\theta)$ with I uniformly chosen at random in $\{1, \ldots, n\}$
- We want to use Monte Carlo samples to approximate $\mathbb{E}X = \nabla f(\theta)$
- We find out C s.t. $\mathbb{E}C$ is easy to compute and such that C highly correlated with X
- Put $Z_\alpha = \alpha(X - C) + \mathbb{E}C$ for $\alpha \in [0, 1]$. We have

$$\mathbb{E}Z_\alpha = \alpha\mathbb{E}X + (1 - \alpha)\mathbb{E}C$$

and

$$\operatorname{var} Z_\alpha = \alpha^2(\operatorname{var} X + \operatorname{var} C - 2\operatorname{cov}(X, C))$$

- Standard variance reduction: $\alpha = 1$, so that $\mathbb{E}Z_\alpha = \mathbb{E}X$ (unbiased)

Variance reduction

Idea: combine SGD with variance reduction

$$\theta^t \leftarrow \theta^{t-1} - \eta\Big(\alpha\big(\nabla f_{i_t}(\theta^{t-1}) - \nabla f_{i_t}(\varphi^{t-1})\big) + \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\varphi^{t-1})\Big)$$

where $\nabla f_i(\varphi^{t-1})$ is the "last computed" gradient of ∇f_i along the iterations

- $\alpha = 1/n$: SAG (Stochastic Average Gradient, Bach et al. 2013)
- $\alpha = 1$: SVRG (Stochastic Variance Reduced Gradient, T. Zhang et al. 2015, 2015)
- $\alpha = 1$: SAGA (Bach et al., 2014)

Variance reduction with SAG

Stochastic Average Gradient (SAG, Bach et al. 2013)

- **Input**: starting point θ_0, learning rate $\eta > 0$
- For $t = 1, 2, \ldots$ until *convergence* do
  - Pick at random (uniformly) i_t in $\{1, \ldots, n\}$
  - Put

$$g_t(i) = \begin{cases} \nabla f_i(\theta^{t-1}) & \text{if } i = i_t \\ g_{t-1}(i) & \text{otherwise} \end{cases}$$

    and compute

$$\theta^t = \theta^{t-1} - \frac{\eta}{n} \sum_{i=1}^{n} g_t(i)$$

- **Return** last θ^t

Variance reduction with SAG

Assume

- Each f_i is L-smooth
- f is μ-strongly convex
- $\eta_t = 1/(16L)$ constant
- Initialize using one epoch of SGD

Non-strongly convex case:

$$\mathbb{E}[f(\theta^t) - f(\theta_*)] \leq O(\frac{\sqrt{n}}{t})$$

Strongly convex case:

$$\mathbb{E}[f(\theta^t) - f(\theta_*)] \leq O\Big(\frac{1}{n\mu} + \frac{L}{n}\Big) \exp\Big(- t\Big(\frac{1}{8n} \wedge \frac{\mu}{16L}\Big)\Big)$$

Improves a lot FG and SGD algorithms

Variance reduction

- Complexity $O(d)$ instead of $O(nd)$ at each iteration
- Choice of a **fixed** step-size $\eta > 0$ possible
- But extra memory required: need to save all the previous gradients.

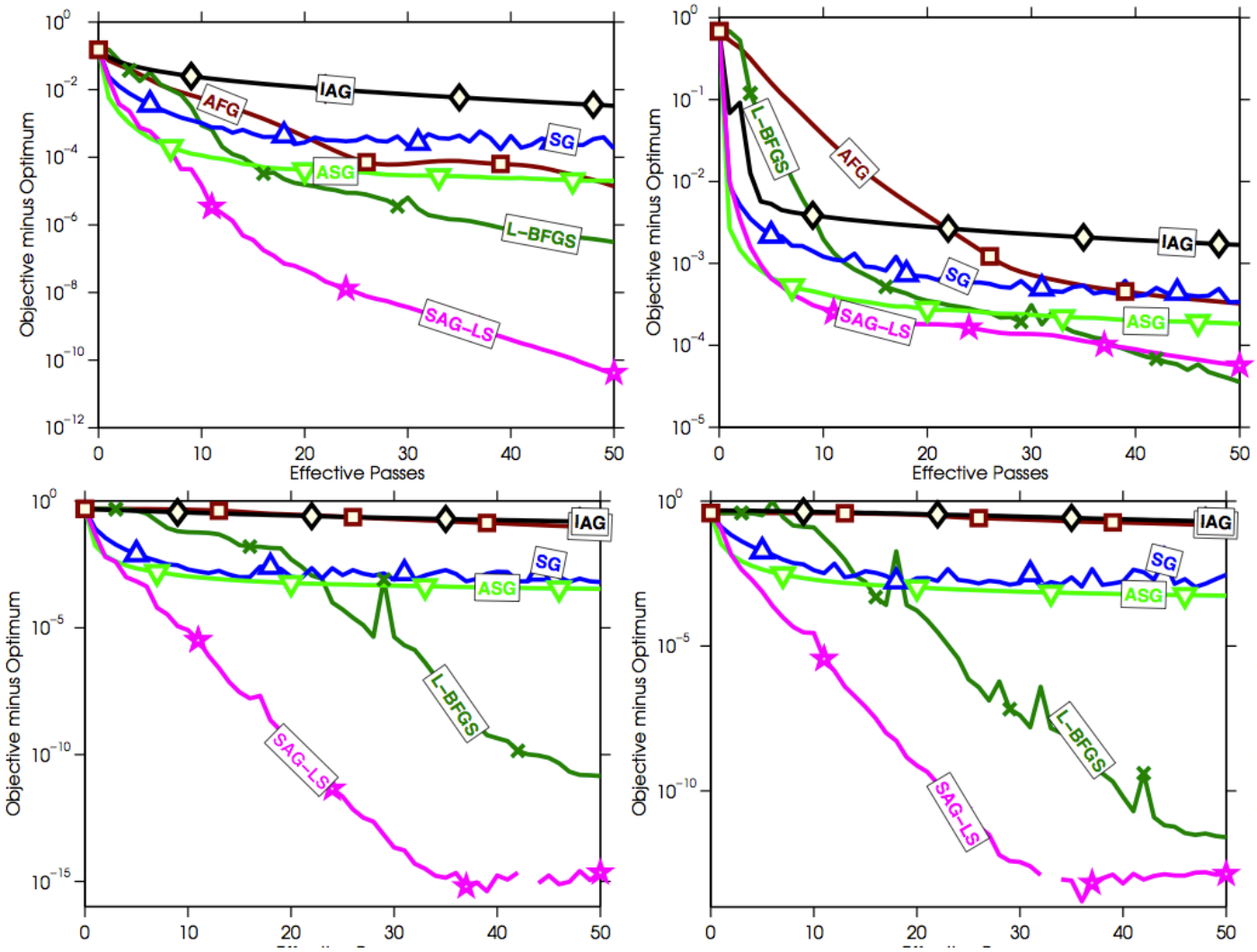
  Hopefully

$$\nabla f_i(\theta) = \ell'(y_i, \langle x_i, \theta \rangle) x_i,$$

  so only need to save $\ell(y_i, \langle x_i, \theta \rangle)$. Memory footprint is $O(n)$ instead of $O(nd)$. If $n = 10^7$, this is 76 Mo

Variance reduction with SAG

Comparison of convergence [Le Roux el al. 2012]



Some practical remarks: file indexing

Now some practical problems / tricks around this

- Need to **index large data files** to be able to read lines at random fast

  Many tools to do this. In `Hadoop`[1] there is the `ArrayFile.Reader` that does that. It's only (roughly) 3x slower than a sequential read of the file.

  Stochastic optimization algorithm also work when using random shuffling of lines at beginning of each epoch: allows to further improve I/O.

---

1 hadoop

http://hadoop.apache.org

Some practical remarks: lazy updates

Feature vectors are usually very sparse (words counts). Complexity of the iteration of a stochastic optimization algorithm can reduced from $O(d)$ to $O(s)$, where s is the sparsity of the features. Important since $d \approx 10^6$ while $s \approx 10^3$

For minimizing

$$\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, \langle \theta, x_i \rangle) + \frac{\lambda}{2}\|\theta\|_2^2$$

an iteration of SGD writes

$$\theta^t = (1 - \eta_t \lambda)\theta^{t-1} - \eta_t \ell'(y_i, \langle x_i, \theta^{t-1} \rangle) x_i$$

If x_i is s sparse, then computing $\eta_t \ell'(y_i, \langle x_i, \theta^{t-1} \rangle) x_i$ is $O(s)$, but $(1 - \eta_t \lambda)\theta t - 1$ is $O(d)$ ...

Some practical remarks: lazy updates

Trick: put $\theta^t = s_t \beta^t$, with $s_t \in [0, 1]$ and $s_t = (1 - \eta_t \lambda) s_{t-1}$

$$\theta^t = (1 - \eta_t \lambda)\theta^{t-1} - \eta_t \ell'(y_i, \langle x_i, \theta^{t-1} \rangle) x_i$$

becomes

$$\begin{aligned} s_t \beta^t &= (1 - \eta_t \lambda) s_{t-1} \beta^{t-1} - \eta_t \ell'(y_i, s_{t-1} \langle x_i, \beta^{t-1} \rangle) x_i \\ &= s_t \beta^{t-1} - \eta_t \ell'(y_i, s_{t-1} \langle x_i, \beta^{t-1} \rangle) x_i \end{aligned}$$

so the iteration is now

$$\beta^t = \beta^{t-1} - \frac{\eta_t}{s_t} \ell'(y_i, s_{t-1} \langle x_i, \beta^{t-1} \rangle) x_i$$

which has complexity $O(s)$.

Some practical remarks: lazy updates

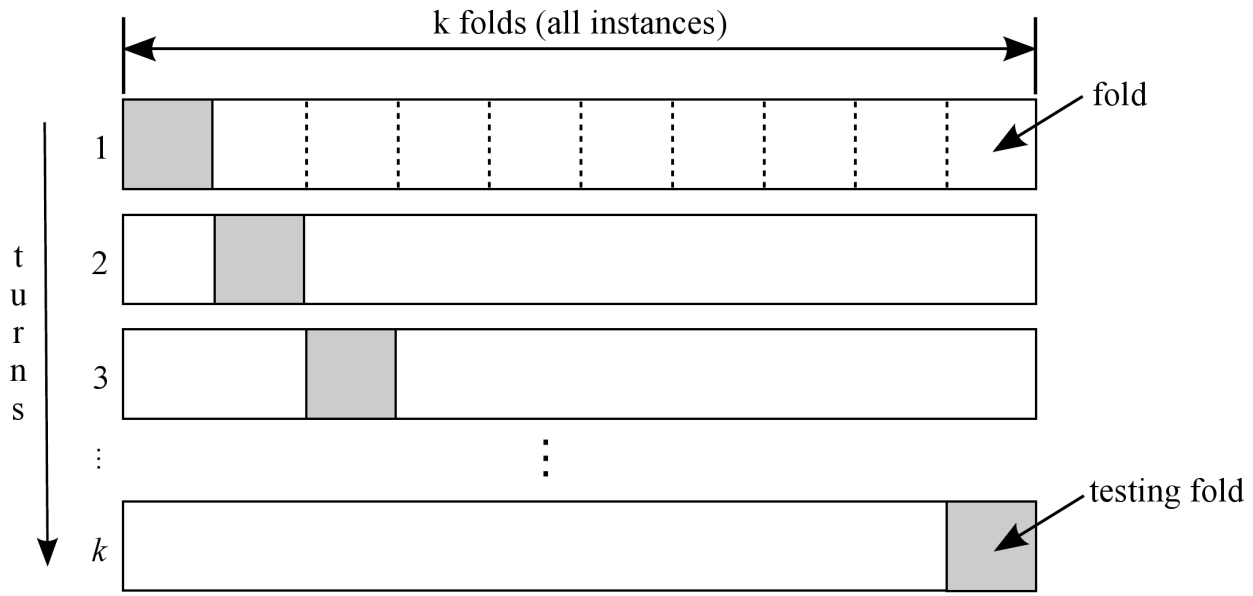
- Just check that s_t is not too small once in a while, in this case put $s_t = 1$ and update θ_t and β_t
- Write the algorithm using only β_t, return $s_t \beta_t$ in the end

Now, complexity of one iteration of a stochastic algorithm is $O(s)$, while an approach based on FG methods is $O(nd)$ without using sparsity

Choice of penalization parameter λ by V-fold with SGD

Quick recap on V-fold:

- Take $V = 5$ or $V = 10$. Pick a random partition $I_1, \ldots, I_V$ of $\{1, \ldots, n\}$, where $|I_v| \approx \frac{n}{V}$ for any $v = 1, \ldots, V$



Some practical remarks: cross-validation

- I don't load the full data in memory
- I can't use V-Fold this way when I'm using an SGD-based solver

Simple solution: when picking a line i at random in the optimization loop, its fold number is given by $i\%V$

- Pick i uniformly at random in $\{1, \ldots, n\}$
- Put $v = i\%V$
- For $v' = 1, \ldots, V$ with $v' \neq v$: Update $\hat{\theta}_{v'}$ using line i
- Update the testing error of $\hat{\theta}_v$ using line i

Cross-validation: warm starting

- So I have many optimization problems to solve for choosing λ!
- If I'm using V-Fold cross-validation, and a choose a set $\Lambda = \{\lambda_1, \ldots, \lambda_M\}$ of values for Λ, it is $V \times |\Lambda|$ problems
- But solutions $\hat{\theta}_{\lambda_{j-1}}$ and $\hat{\theta}_{\lambda_j}$ are going to be close when λ_{j-1} and λ_j are

Use **warm starts**:

- Fix parameters $\lambda_1 < \lambda_2 < \cdots < \lambda_M$
- Put $\theta_0 = 0$ (I don't know where to start)
- For $m = M, \ldots, 1$
  - Put $\lambda = \lambda_m$
  - Solve the problems starting at θ_0 for this value of λ (on each fold)
  - Keep the solutions $\hat{\theta}$ (test it, save it...)
  - Put $\theta_0 \leftarrow \hat{\theta}$

This allows to solve much more rapidly the sequence of problems

Collaborative Filtering

What is collaborative filtering?

Many **users**

Many **items**

- Based on many observed user-item interactions (rating, purchase, click)
- Predict new interactions
- Does Bob like strawberries?

Collaborative Filtering

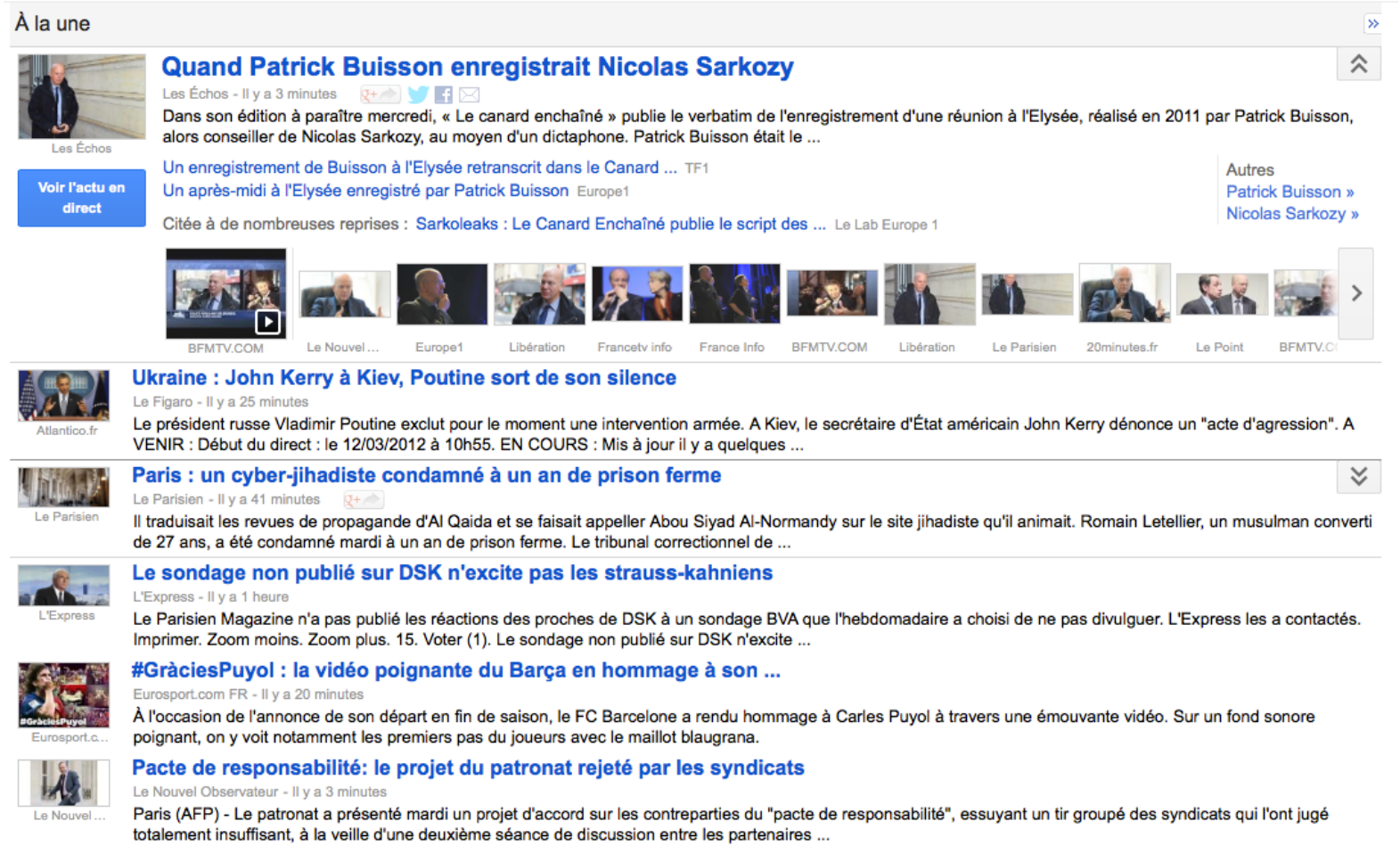
Amazon.com recommends products based on your purchases, browsing history

- but based on the purchases and history of other users too

Collaborative Filtering

Google News recommends news based on your browsing activity

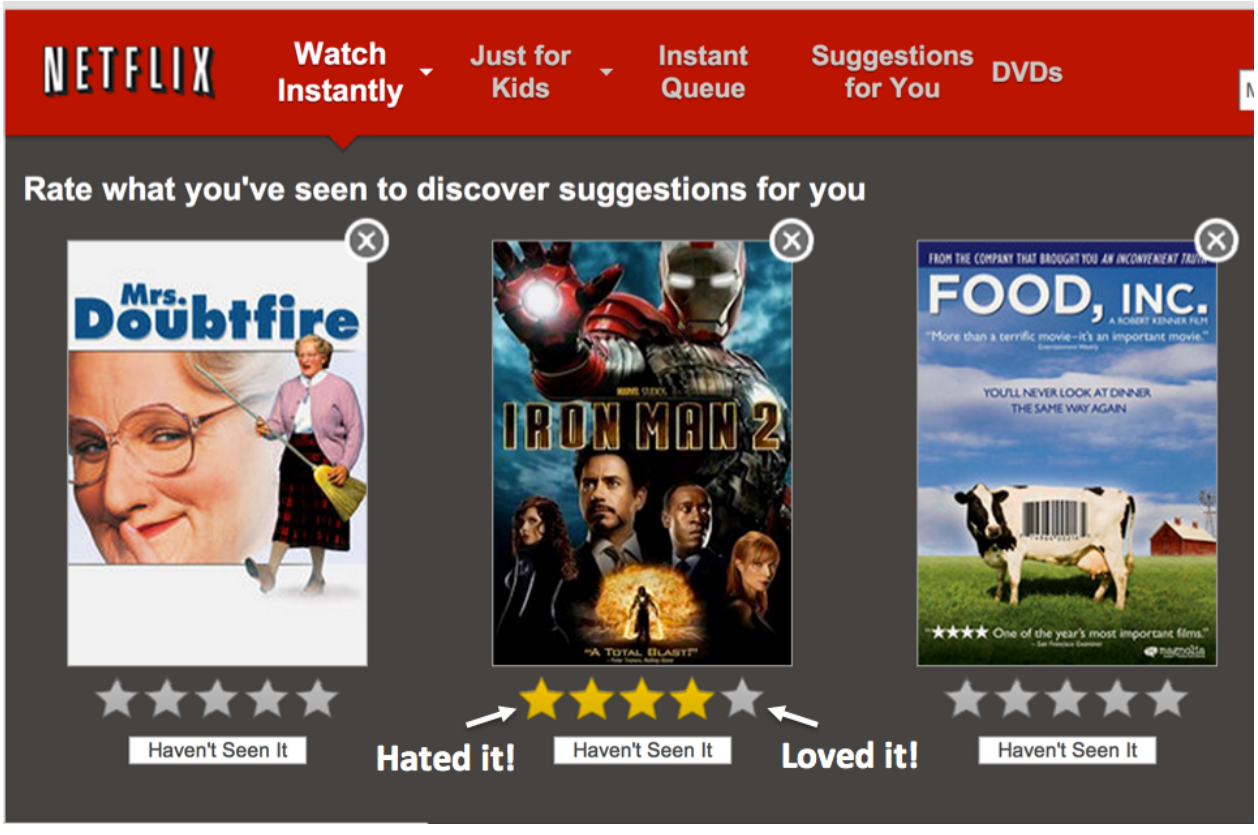
- but on the browsing activity of other users too



Collaborative Filtering

Netflix predicts the rating you'd give to a movie

- using all the ratings given by all users



- 60% of Netflix's DVD rentals due to recommandations

Collaborative Filtering

Netflix predicts the rating you'd give to a movie

- using all the ratings given by all users

- 60% of Netflix's DVD rentals due to recommandations

Collaborative Filtering: Netflix

Collaborative Filtering: Netflix Prize

Collaborative Filtering: Netflix Prize

Collaborative Filtering: Netflix Prize

Netflix Prize

- October 2, 2006
- Dataset: 100 million ratings , n_U =480K users, n_I =18K movies
- **only 1.1%** of the matrix is filled!
- **Goal**: create a computer code that predicts ratings
- $1m grand prize for anyone beating **Cinematch** accuracy by 10%
- 5000 teams over 150 countries participated

Collaborative Filtering, Matrix completion: **fill unobserved entries** of a matrix

- Unknown matrix M has size $n_U \times n_I$
- Observe $m \ll n_U n_I$ entries (100m $\ll$ 8.64M for Netflix)
- Impossible!

Collaborative Filtering

There is hope:

- Personal preferences are correlated: if Alice likes A and B and Bob likes A, B and C, then Alice is more likely to like C
- There are latent factors that describe the data in a much lower dimensional space. Groups of users, groups of movies, factors that explain the taste of users. High-dimensionality but hidden low-dimensional structure
- The hidden matrix is (approximately) **low-rank**

Collaborative Filtering task:

- discover patterns (low-dimensional hidden structure, latent factors)
- use these patterns for prediction of new user-item interactions
- Do not necessarily use item or user attributes (demographic information, author, genre, cast, plot, etc.)

Given:

- Users $u \in \{1, \ldots, n_U\}$
- Items $i \in \{1, \ldots, n_I\}$
- When u has an interaction with item i, (watches the movie, clicks on a banner, buys a product), he enters a scalar rating $r_{u,i}$ (number of clicks, rating of a movie, etc.)
- Set E of pairs (u, i) of observed ratings $r_{u,i}$

Matrix completion problem

$$M = \begin{bmatrix} ? & ? & 2 & \cdots & 5 \\ 2 & ? & 1 & \cdots & ? \\ ? & 2 & ? & \cdots & 4 \end{bmatrix}$$

Measures of success. Decompose $E = E_{\text{train}} \cup E_{\text{test}}$ into training and testing respectively. $r_{u,i}$ = ground truth and $\hat{r}_{u,i}$ = estimated rating

- Root Mean Square Error

$$\texttt{RMSE} = \sqrt{\frac{1}{|E_{\text{test}}|} \sum_{(u,i) \in E_{\text{test}}} (r_{u,i} - \hat{r}_{u,i})^2}$$

- Mean Absolute Error

$$\texttt{MAE} = \frac{1}{|E_{\text{test}}|} \sum_{(u,i) \in E_{\text{test}}} |r_{u,i} - \hat{r}_{u,i}|$$

- Ranking error: fraction of true top-5 preferences are in my predicted top 5?

Biases

- Remove biases from the ratings
- Some users give systematically higher ratings
- Some items gets systematically better rates (old movies...)
- Don't forget that PCA needs centering

Remove bias

$$\widetilde{r}_{u,i} = r_{u,i} - b_{u,i}$$

Let's denote

- $U(i)$ the set of users who rated item i
- $I(u)$ the set of items rated by user u

Biases

Let's compute means! We can choose $b_{u,i}$ as one of the following:

- Global mean

$$b = \frac{1}{|E|} \sum_{(u,i) \in E} r_{u,i}$$

- Item's mean rating

$$b_i = \frac{1}{|U(i)|} \sum_{u \in U(i)} r_{u,i}$$

- User's mean rating

$$b_u = \frac{1}{|I(u)|} \sum_{i \in I(u)} r_{u,i}$$

- Item's mean rating + user's mean deviation from item mean

$$b_{u,i} = b_i + \frac{1}{|I(u)|} \sum_{i' \in I(u)} (r_{u,i'} - b_{i'})$$

Interpolating means

- Some users have much more ratings than other (1000 $\times$ more!)
- Users with a small numbers of ratings are not as reliable as ones: more noisy. It's hard to trust a mean with only one rating!
- Interpolate between a global estimate and an estimate from user's data

Interpolate to have a better bias estimation:

$$\widetilde{b}_u = \frac{\alpha}{\alpha + |I(u)|} b + \frac{|I(u)|}{\alpha + |I(u)|} b_u$$

where we recall that b is the global mean

$$b = \frac{1}{|E|} \sum_{(u,i) \in E} r_{u,i}$$

and b_u is the user's mean

$$b_u = \frac{1}{|I(u)|} \sum_{} r_{u,i}$$

CF as classification / regression

CF is a set of n_I classifications / regression problem: one for each $i \in \{1, \ldots, n_I\}$

- Consider a fixed i
- Treat each user as a vector (with many missing values) or ratings of all item except i
- The class of each user with respect to i is given by the user's rating
- Prediction of the rating $\hat{r}_{u,i}$ = classication of a user's vector $r_{u,i}$
- Reduces CF to the well-known classification problem
- But, with a huge number of classes!
- This approach does not take advantage of the problem structure

K-NN

K-Nearest Neighbor Method (K-NN) most widely used family of methods

- item-based or user-based
- for product recommandation: **item-based**
- represent each item as a vector of user's ratings with many missing values $r_{\bullet,i} = [2, ?, 1, ?, ?, ?, 2, 1, ?, ?, ?, 5]$

Idea: users rate similar items similarly. In order to predict a rating $r_{u,i}$ for user u and item i:

- Compute similarity between i and every other items
- Find the K items rated by u most similar to i
- Compute weighted average of these ratings

K-NN: Similarity measures

How to measure similarity between items?

- Cosine similarity

$$d(r_{\bullet,i}, r_{\bullet,i'}) = \frac{\langle r_{\bullet,i}, r_{\bullet,i'} \rangle}{\|r_{\bullet,i}\| \|r_{\bullet,i'}\|}$$

- Pearson correlation coefficient

$$d(r_{\bullet,i}, r_{\bullet,i'}) = \frac{\langle r_{\bullet,i} - \bar{r}_{\bullet,i}, r_{\bullet,i'} - \bar{r}_{\bullet,i'} \rangle}{\|r_{\bullet,i} - \bar{r}_{\bullet,i}\| \|r_{\bullet,i'} - \bar{r}_{\bullet,i'}\|}$$

- Inverse Euclidean distance

$$d(r_{\bullet,i}, r_{\bullet,i'}) = \frac{1}{\|r_{\bullet,i} - r_{\bullet,i'}\|}$$

Vectors $r_{\bullet,i}$ contains many missing values: compute these similarities over subets of users that rated both items i and i'

K-NN

How to choose the K-nearest neighbors?

- Select the K items with largest similarity score to item i, among the items rated by u, denoted $N(i, u)$
- Prediction given by

$$\hat{r}_{u,i} = b_{u,i} + \sum_{i' \in N(i,u)} w_{i,j}(r_{u,i'} - b_{u,i'})$$

where $w_{i,i'}$ weights

Example of weights:

- Equal weights

$$w_{i,i'} = \frac{1}{N(u,i)}$$

- Similarity weights

$$w_{i,i'} = \frac{d(r_{\bullet,i}, r_{\bullet,i'})}{\sum_{i'' \in N(i,u)} d(r_{\bullet,i}, r_{\bullet,i''})}$$

Event better: user optimized weights.

- Choose weights that best predict other known ratings of i among all users that rated i
- Corresponds to many small linear regression problems
- Needs to store **many** weights $O(n_I^2)$

Conclusion for K-NN methods

- Easy to implement
- No training time
- Flexible
- But need to store many parameters (all item, vectors, weights in memory)
- Don't exploit hidden low-dimensional structure

MF: if I know the item's features

Matrix Factorization (= MF)

- Assume that we know features about the items

$$y_i = [\text{cast}, \text{year}, \text{genre}, \cdots] \in \mathbb{R}^r$$

  for all $i = 1, \ldots, n_I$.
- r features for each item
- We want the users parameters x_u for $u = 1, \ldots, n_U$

Linear regression

$$\hat{x}_u \in \operatorname*{argmin}_{x_u} \sum_{i \in I(u)} (r_{u,i} - \langle x_u, y_i \rangle)^2.$$

Even better: **ridge regression**

$$\hat{x}_u = \operatorname*{argmin}_{x_u} \sum_{i \in I(u)} (r_{u,i} - \langle x_u, y_i \rangle)^2 + \lambda \|x_u\|_2^2$$

- But we don't want to construct ad-hoc features y_i for items
- Not a good idea for building recommandations

MF: if I know the user's features

- Assume that we know user's features x_u for all $u = 1, \ldots, n_U$.
- r features for each user
- We want the items features y_i for $i = 1, \ldots, n_I$

Once again: **linear regression**

$$\hat{y}_i \in \operatorname*{argmin}_{y_i} \sum_{u \in U(i)} (r_{u,i} - \langle x_u, y_i \rangle)^2.$$

Even better: **ridge regression**

$$\hat{y}_i \in \operatorname*{argmin}_{y_i} \sum_{u \in U(i)} (r_{u,i} - \langle x_u, y_i \rangle)^2 + \lambda \|y_i\|_2^2$$

- But we can't construct ad-hoc features x_u for users
- Still not a good idea for building recommandations

MF: put everything together

- We don't want to construct ad-hoc features y_i for items
- We can't construct ad-hoc features x_u for users

So let's learn items and users features **at the same time**!

- Putting things together

$$\hat{x}_u = \underset{x_u}{\operatorname{argmin}} \sum_{i \in I(u)} (r_{u,i} - \langle x_u, \hat{y}_i \rangle)^2 + \lambda \|x_u\|_2^2$$

$$\hat{y}_i = \underset{y_i}{\operatorname{argmin}} \sum_{u \in U(i)} (r_{u,i} - \langle \hat{x}_u, y_i \rangle)^2 + \lambda \|y_i\|_2^2$$

for all $u = 1, \ldots, n_U$ and $i = 1, \ldots, n_I$

MF: put everything together

$$\hat{x}_u = \underset{x_u}{\operatorname{argmin}} \sum_{i \in I(u)} (r_{u,i} - \langle x_u, \hat{y}_i \rangle)^2 + \lambda \|x_u\|_2^2$$

$$\hat{y}_i = \underset{y_i}{\operatorname{argmin}} \sum_{u \in U(i)} (r_{u,i} - \langle \hat{x}_u, y_i \rangle)^2 + \lambda \|y_i\|_2^2$$

for all $u = 1, \ldots, n_U$ and $i = 1, \ldots, n_I$

- Hum... the $\hat{x}_u$'s depends on the $\hat{y}_i$'s that depend on the $\hat{x}_u$'s that depends on the $\hat{y}_i$ that... !

MF: put everything together

Let's rewrite this. Put

$$X^\top = \begin{bmatrix} \vdots & \vdots & \vdots \\ x_1 & \cdots & x_{n_U} \\ \vdots & \vdots & \vdots \end{bmatrix} \quad \text{and} \quad Y^\top = \begin{bmatrix} \vdots & \vdots & \vdots \\ y_1 & \cdots & y_{n_I} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Then we consider the minimization of

$$F(X, Y) = \sum_{(u,i)\in E} (r_{u,i} - \langle x_u, y_i \rangle)^2 + \lambda \sum_{u=1}^{n_U} \|x_u\|_2^2 + \lambda \sum_{i=1}^{n_I} \|y_i\|_2^2$$

over $X \in \mathbb{R}^{n_U \times r}$ and $Y \in \mathbb{R}^{n_I \times r}$ jointly.

- The penalization terms $\lambda \sum_{u=1}^{n_U} \|x_u\|_2^2 + \lambda \sum_{i=1}^{n_I} \|y_i\|_2^2$ counters overfitting

$$F(X,Y) = \sum_{(u,i)\in E} (r_{u,i} - \langle x_u, y_i \rangle)^2 + \lambda \sum_{u=1}^{n_U} \|x_u\|_2^2 + \lambda \sum_{i=1}^{n_I} \|y_i\|_2^2$$

over $X \in \mathbb{R}^{n_U \times r}$ and $Y \in \mathbb{R}^{n_I \times r}$ jointly.
Let's write this matricially:

$$F(X,Y) = \|\mathcal{P}_E(R - XY^\top)\|_F^2 + \lambda \|X\|_F^2 + \lambda \|Y\|_F^2$$

where

$$\|A\|_F = \text{ Frobenius norm of } A = \sqrt{\sum_{j,k} A_{j,k}^2}$$

and

$$\mathcal{P}_E(A) = \begin{cases} A_{u,i} & \text{if } (u,i) \in E \\ 0 & \text{otherwise} \end{cases}$$

Put $\lambda = 0$ and $E = \{1, \ldots, n_U\} \times \{1, \ldots, n_I\}$

$$F(X, Y) = \|\mathcal{P}_E(R - XY^\top)\|_F^2 = \|R - XY^\top\|_F^2$$

Solution is given by the SVD (Singular Value Decomposition)
Recall that

- X of size $n_U \times r$
- Y of size $n_I \times r$

Then

$$\underset{X,Y}{\operatorname{argmin}} \|R - XY^\top\|_F^2$$

is given by thresholded SVD of R

SVD (Singular Value Decomposition)

Any matrix $R \in \mathbb{R}^{n_U \times n_I}$ writes

$$R = U \Sigma V^\top$$

where

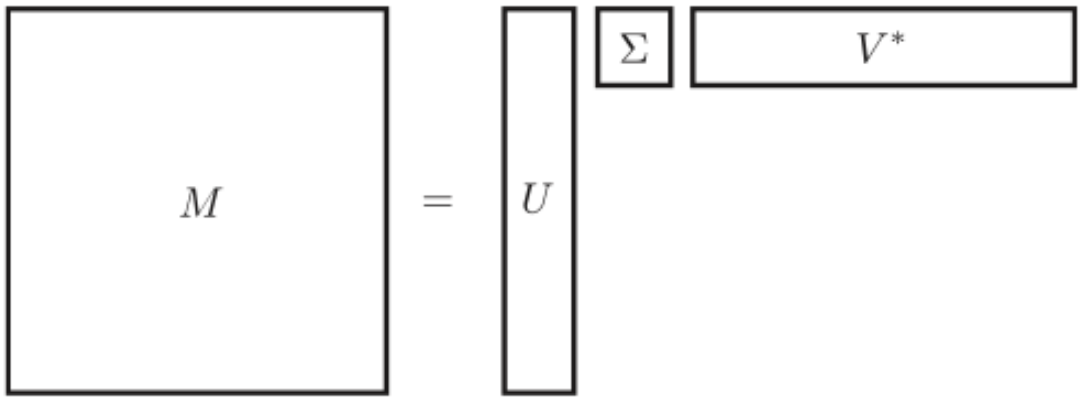
- U is the matrix of **left singular vectors** (columns of U are eigenvectors of $RR^\top$, it satisfies $U^\top U = I$
- V is the matrix of **right singular vectors** (eigenvectors of $R^\top R$, it satisfies $V^\top V = I$
- $\Sigma = \text{diag}[\sigma_1, \ldots, \sigma_{n_U \wedge n_I}]$ is the diagonal matrix containing the **singular values**

  $$\sigma_1 \geq \cdots \geq \sigma_{n_U \wedge n_I}$$

  where

  $$\sigma_j(X) = \sqrt{\lambda_j(X^\top X)} = j\text{th eigenvalue of } X^\top X$$

Matrix Factorization: SVD



Fundamental result:

$$\underset{M \in \mathbb{R}^{n_U \times n_I} : \text{rank}(X) = r}{\text{argmin}} \|R - M\|_2^2 = U_r \Sigma_r V_r^\top$$

where $R = U \Sigma V^\top$ is the SVD of R and

- $\Sigma_r = \text{diag}[\sigma_1, \ldots \sigma_r]$
- U_r contains the first r columns of U
- V_r contains the first r columns of V

- Don't forget that PCA = SVD of the covariance matrix

Matrix Factorization: SVD

Hence a solution of

$$(\hat{X}, \hat{Y}) \in \underset{X,Y}{\operatorname{argmin}} \|R - XY^\top\|_F^2$$

is given by

$$\hat{X} = U_r \Sigma_r \quad \text{and} \quad \hat{Y} = V_r^\top$$

- Matrix completion can be understood as an SVD with missing entries
- With extra regularization to avoid overfitting using ridge penalization

Ok. So how do I solve

$$F(X,Y) = \sum_{(u,i)\in E} (r_{u,i} - \langle x_u, y_i \rangle)^2 + \lambda \sum_{u=1}^{n_U} \|x_u\|_2^2 + \lambda \sum_{i=1}^{n_I} \|y_i\|_2^2$$

or equivalently

$$F(X,Y) = \|\mathcal{P}_E(R - XY^\top)\|_F^2 + \lambda \|X\|_F^2 + \lambda \|Y\|_F^2$$

???

Algorithm 1. Alternating Least-Squares (ALS)

Idea: if we knew Y we could solve X using ridge regression and vice-versa: alternate between optimizing on X and Y with the other matrix fixed

Alternating least-squares (ALS) algorithm

Repeat until *convergence*:

- For each u solve the linear system:
  $x_u^{\text{new}} \leftarrow$ solution of $\sum_{i \in I(u)} (y_i y_i^\top + \lambda I) x_u = \sum_{i \in I(u)} r_{u,i} y_i$
- For each item i solve
  $y_i^{\text{new}} \leftarrow$ solution of $\sum_{u \in U(i)} (x_u x_u^\top + \lambda I) y_i = \sum_{u \in U(i)} r_{u,i} x_u$
- $x_u \leftarrow x_u^{\text{new}}$, $y_i \leftarrow y_i^{\text{new}}$

- Updates for x_u and y_i can be done in parallel
- Complexity. Space: $O(n_u r + n_I r)$ and time: $O(n_u r^3 + n_I r^3)$ per iteration. $O(r^3)$ for solving the linear systems
- No need to store the complete ratings matrix

Algorithm 2. Gradient Descent (GD)

Idea: use standard gradient descent

- $\nabla_{x_u} F(X, Y) = \lambda x_u + \sum_{i \in I(u)} (\langle x_u, y_i \rangle - r_{u,i}) y_i$
- $\nabla_{y_i} F(X, Y) = \lambda y_i + \sum_{u \in U(i)} (\langle x_u, y_i \rangle - r_{u,i}) x_u$

Gradient Descent algorithm

Repeat until *convergence*:

- For each u update
  $x_u^{\text{new}} \leftarrow x_u - \eta \left(\lambda x_u + \sum_{i \in I(u)} (\langle x_u, y_i \rangle - r_{u,i}) y_i \right)$
- For each i update
  $y_i^{\text{new}} \leftarrow y_i - \eta \left(\lambda y_i + \sum_{u \in U(i)} (\langle x_u, y_i \rangle - r_{u,i}) x_u \right)$
- $x_u \leftarrow x_u^{\text{new}}$, $y_i \leftarrow y_i^{\text{new}}$

- Updates for x_u and y_i can be done in parallel
- Complexity: $O(n_u r + n_I r)$ no $O(r^3)$ overhead iteration
- No need to store the complete ratings matrix

Algorithm 3. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent algorithm

Repeat until *convergence*:

- Choose $(u, i) \in E$ at random
- Update x_u
  $x_u^{\text{new}} \leftarrow x_u - \eta(\lambda x_u + (\langle x_u, y_i \rangle - r_{u,i})y_i)$
- Update y_i
  $y_i^{\text{new}} \leftarrow y_i - \eta(\lambda y_i + (\langle x_u, y_i \rangle - r_{u,i})x_u)$
- $x_u \leftarrow x_u^{\text{new}}$, $y_i \leftarrow y_i^{\text{new}}$

- Complexity: $O(n_u r + n_I r)$ no $O(r^3)$ overhead iteration
- No need to store the complete ratings matrix

Conclusion for CF using Matrix Factorization

Parameters to tune:

- step-size, or learning rate η. Must be decreasing for SGD
- Regularization parameter $\lambda > 0$ and number of latent factors r. Tuned using cross-validation

There is a big problem:

$$F(X, Y) = \sum_{(u,i)\in E} (r_{u,i} - \langle x_u, y_i \rangle)^2 + \lambda \sum_{u=1}^{n_U} \|x_u\|_2^2 + \lambda \sum_{i=1}^{n_I} \|y_i\|_2^2$$

is **not** a convex problem

- Local minimum, initialization is important
- No guarantees towards a good minimum
- Mostly heuristics

A convex formulation of the matrix completion problem

- Unknown matrix R of size $n_U \times n_I$
- If R has rank r, its degrees of freedom are $r(n_U + n_I - r)$
- $E \subset \{1, \dots, n_U\} \times \{1, \dots, n_I\}$ of observed entries of R
- We need $|E| \geq r(n_U + n_I - r)$ (otherwise no hope to recover R
- We observe only $\mathcal{P}_E(R)$

We assume that the rank of R is small. So let's penalize the rank

- Tempting to consider

$$\hat{R} \in \operatorname*{argmin}_{M \in \mathbb{R}^{n_U \times n_I}} \left\{ \frac{1}{2} \|\mathcal{P}_E(M - R)\|_F^2 + \lambda \operatorname{rank}(M) \right\}$$

- Too hard
- For Lasso we've found that a convex relaxation of ℓ_0 is ℓ_1
- Can't we do the same for the rank?

Yes!

A convex formulation of the matrix completion problem

$$\text{rank}(M) = \sum_{k=1}^{n_U \wedge n_I} \mathbf{1}_{\sigma_j(M)>0} = \|\sigma(M)\|_0$$

Replace ℓ_0 by ℓ_1:

$$\|M\|_* = \sum_{j=1}^{n_I \wedge n_U} \sigma_j(M)$$

Hence tempting to consider

$$\hat{R} \in \underset{M \in \mathbb{R}^{n_I \times n_U}}{\text{argmin}} \left\{ \frac{1}{2} \|\mathcal{P}_E(M - R)\|_2^2 + \lambda \|M\|_* \right\}$$

for a regularization parameter $\lambda > 0$. This is a **convex** problem!

Proximal gradient descent

Proximal gradient descent for the CF problem

Repeat until *convergence*:

- $M \leftarrow S_{\lambda\eta_k}(M - \eta_k(\mathcal{P}_E(M - R)))$

where S_λ is the spectral soft-thresholding operator: if $M = U\Sigma V^\top$ SVD of M, then

$$S_\lambda(M) = U\,\mathrm{diag}[(\sigma_1 - \lambda)_+, \ldots, (\sigma_{n_1 \wedge n_2} - \lambda)_+]V^\top$$

Thresholding of the singular value: the solution will be of low rank. Many other algorithms, more memory efficient and faster

Proximal gradient descent

- Convex problem: convex optimization and convergence guarantees to a minimum
- Bottleneck: an SVD is necessary at **each iteration**! Complexity of an SVD $O((n_U \vee n_I)(n_U \wedge n_I)^2)$
- Can be reduced using partial SVD (compute only k top singular values and vectors). Complexity is (best case) $O(n_1 n_2 k)$ [keyword: **Lanczos** algorithms]
- Compute an approximate solution, given some tolerance
- For remedy for large SVD is the **divide and conquer** principle

Sketch of application: image inpainting

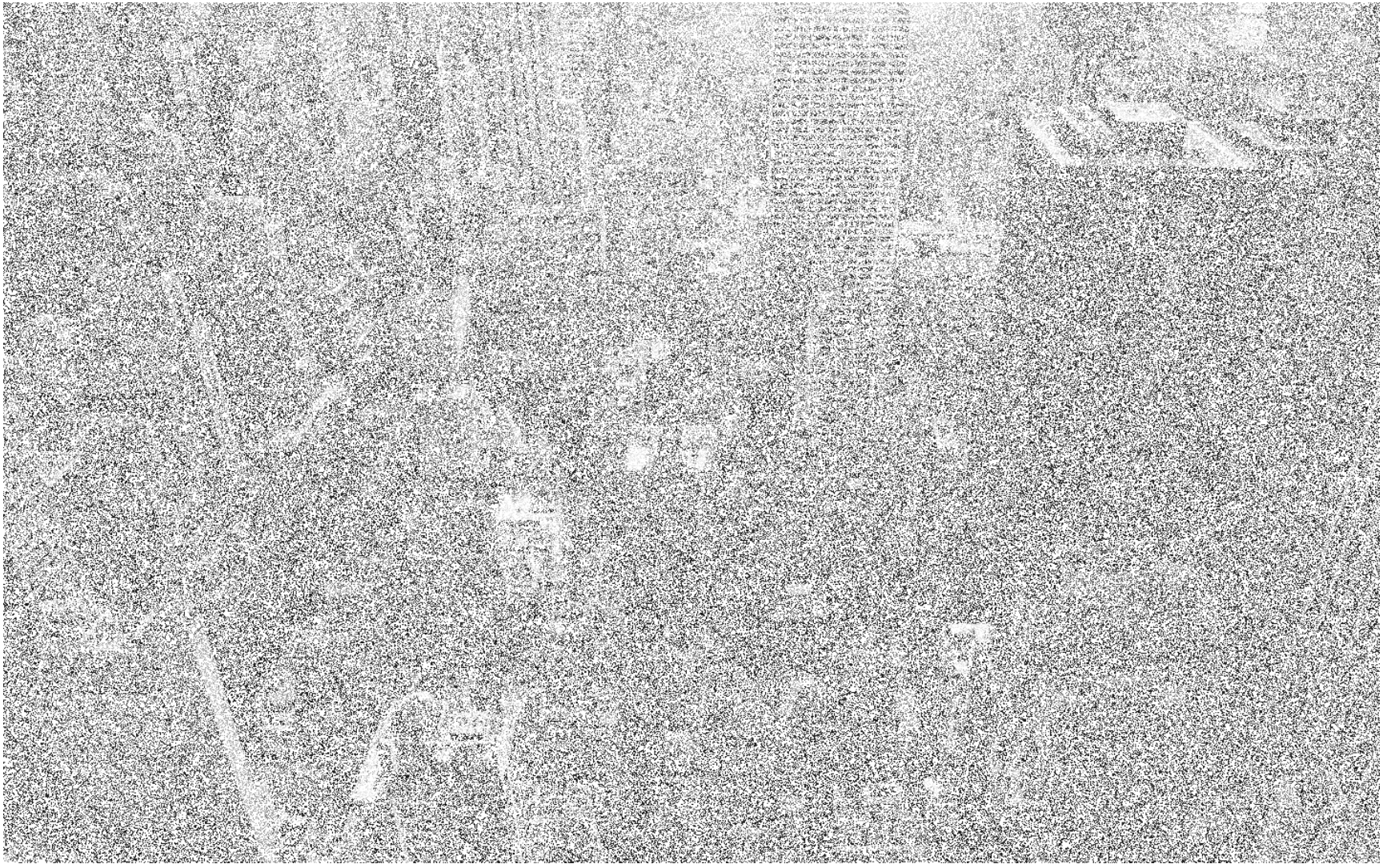
Sketch of application: image inpainting

Sketch of application: image inpainting

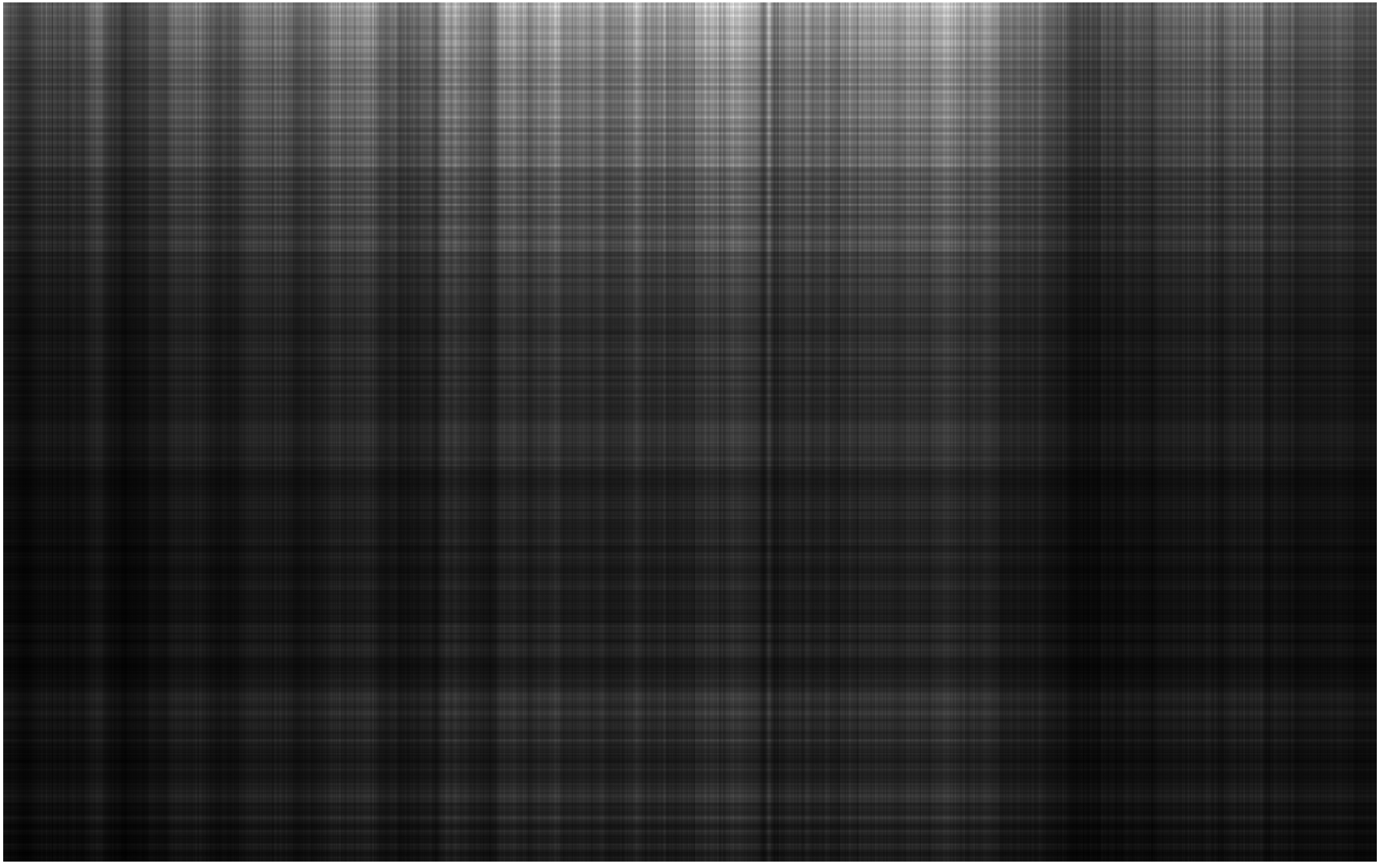
Sketch of application: image inpainting

Sketch of application: matrix completion

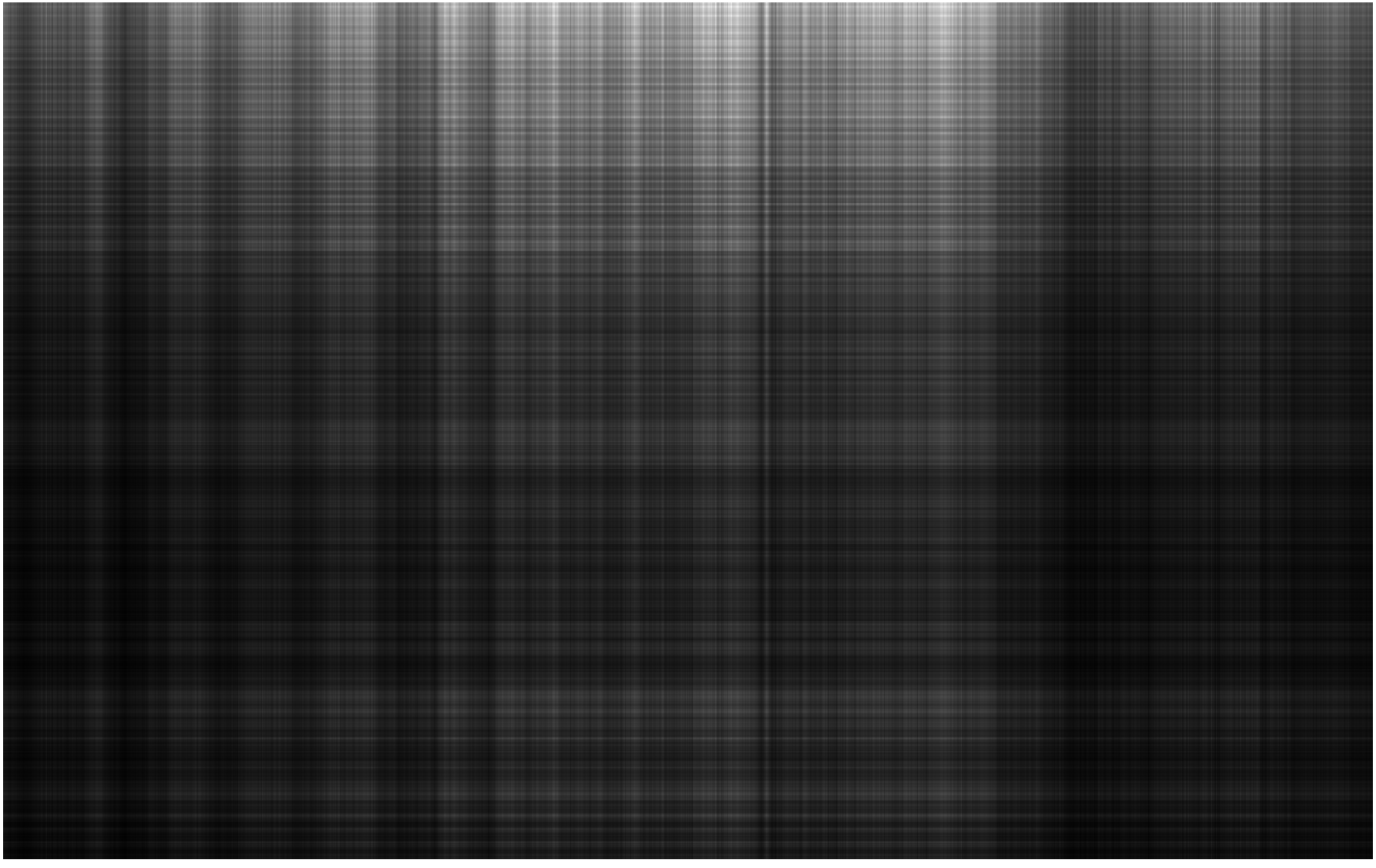
We only observe 35% of the picture



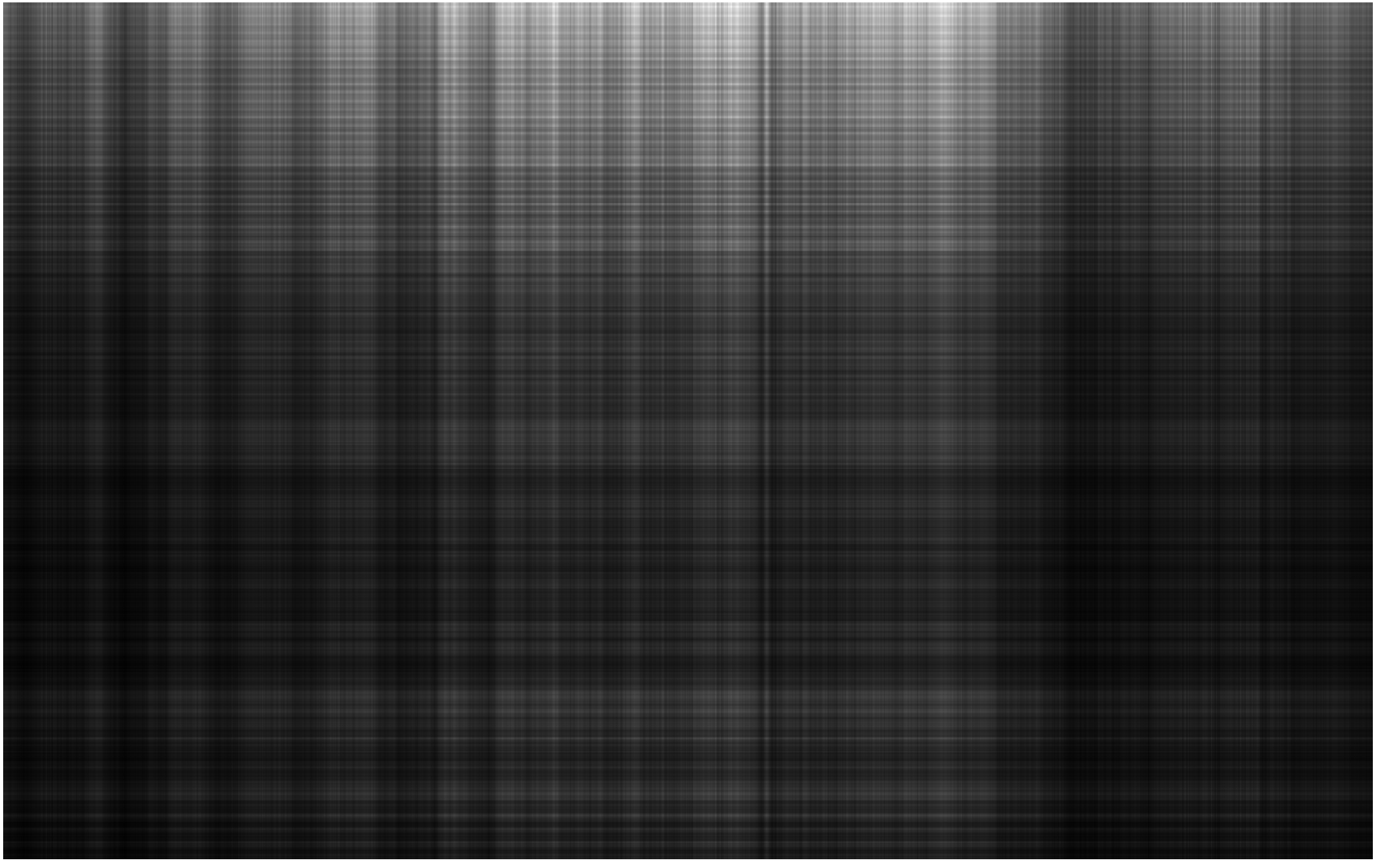
Collaborative Filtering, Matrix Completion



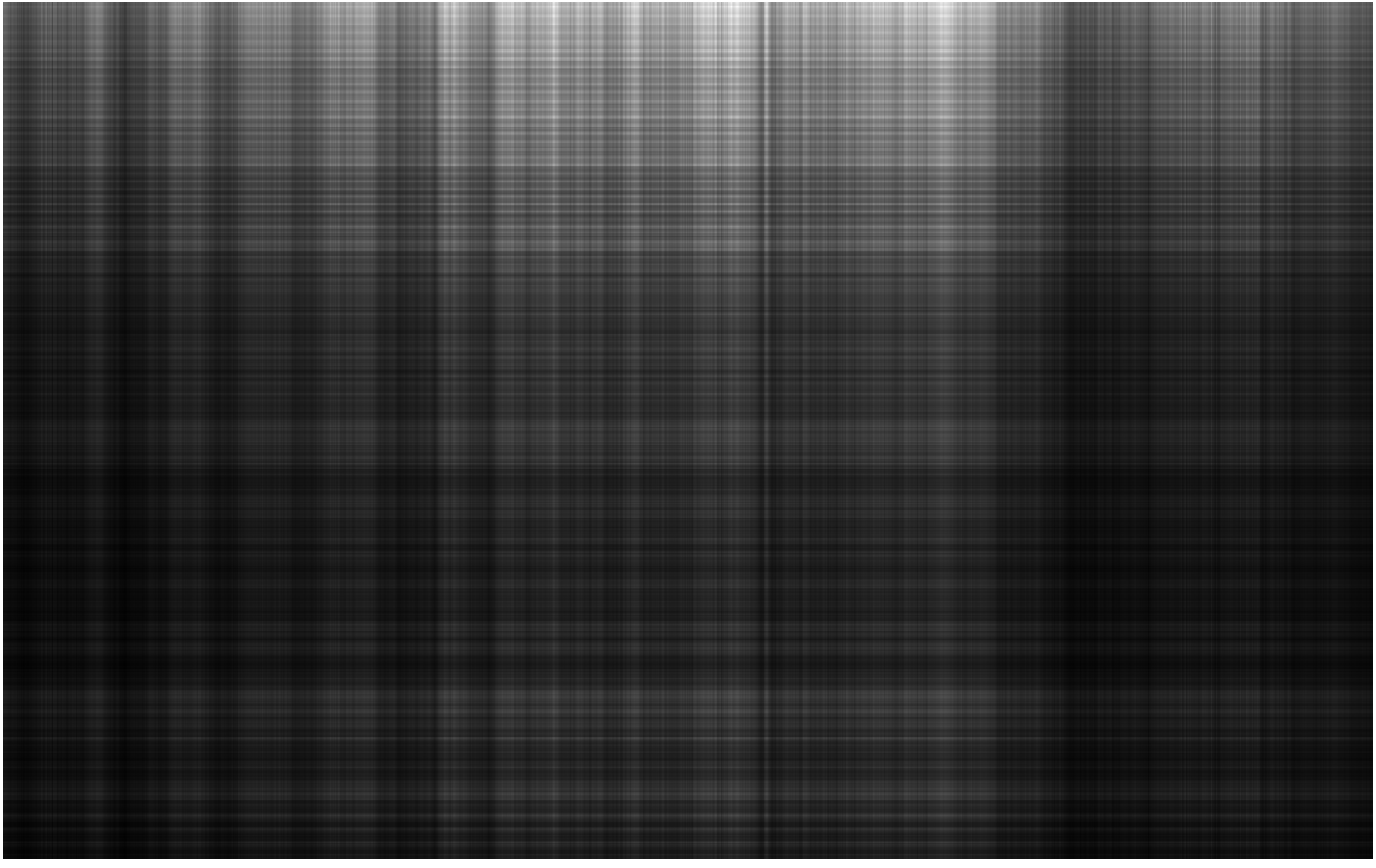
Collaborative Filtering, Matrix Completion



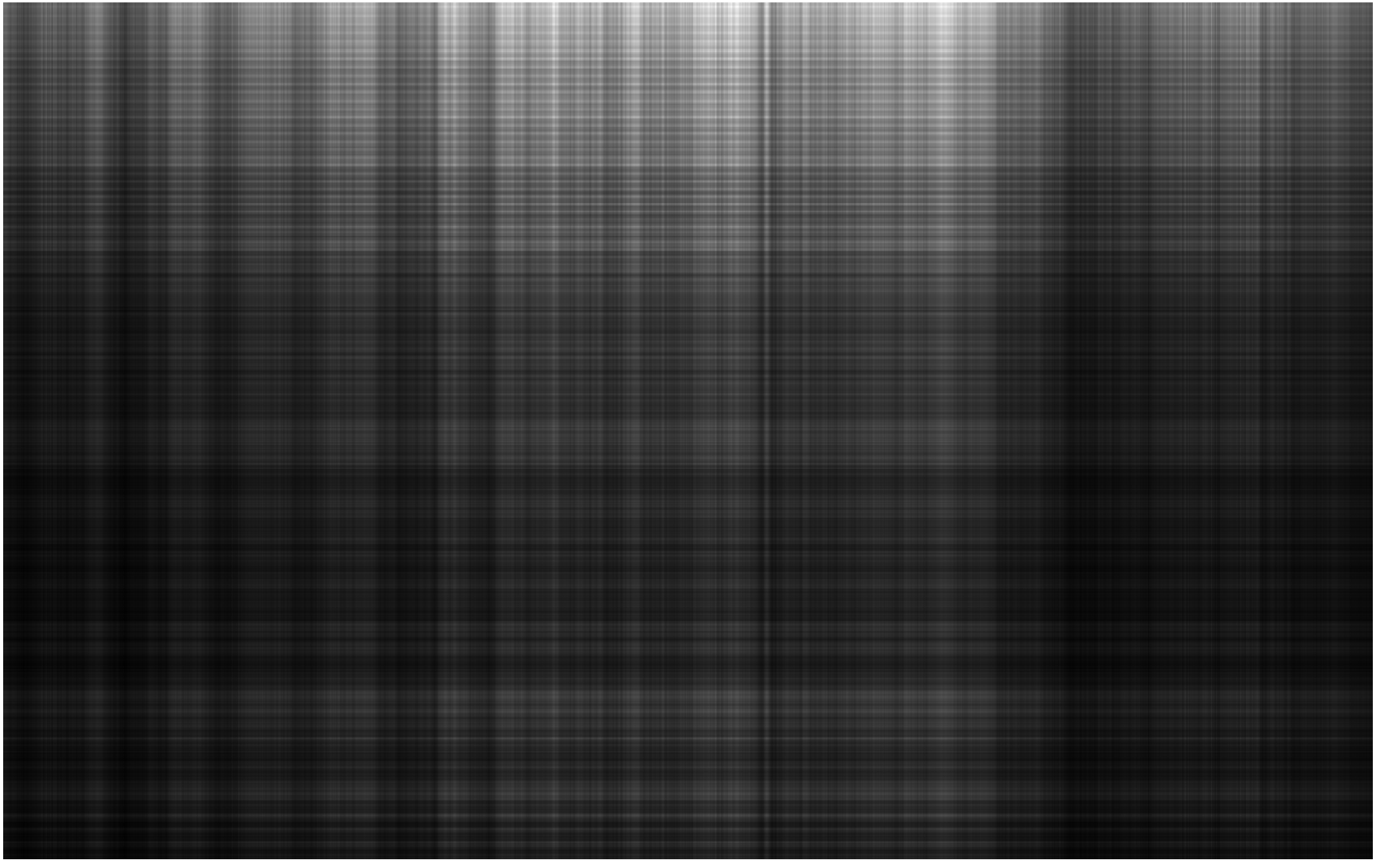
Collaborative Filtering, Matrix Completion



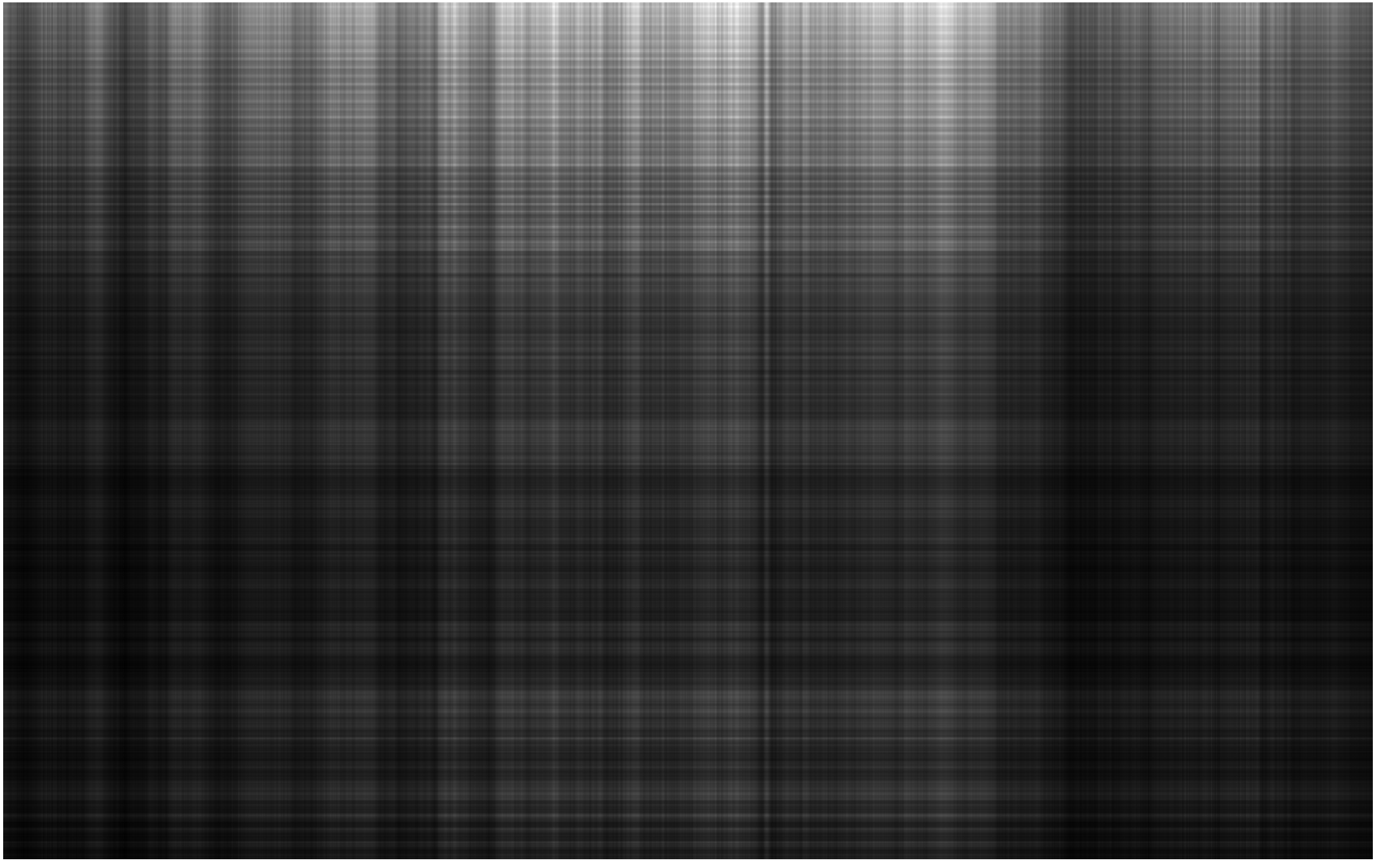
Collaborative Filtering, Matrix Completion



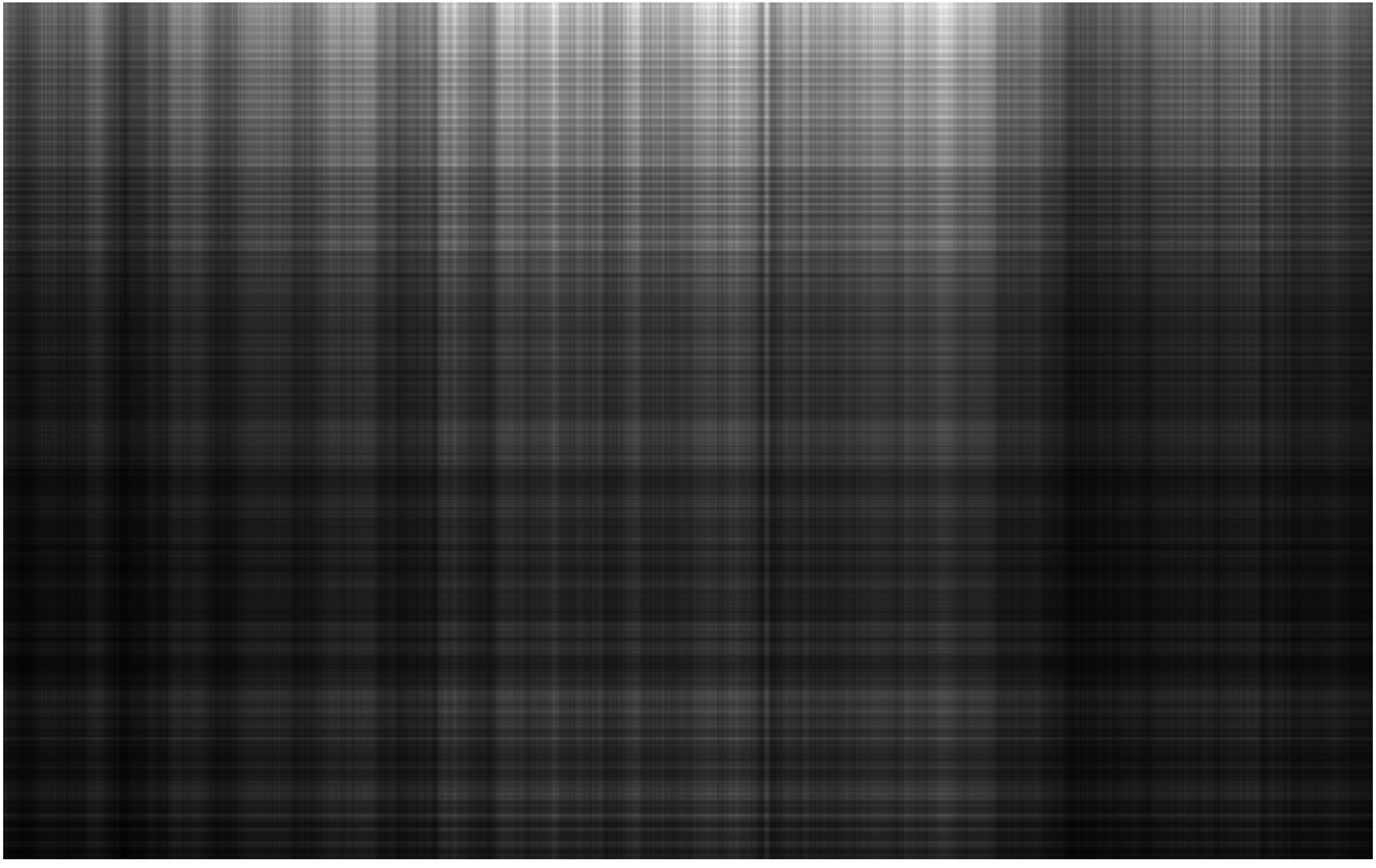
Collaborative Filtering, Matrix Completion



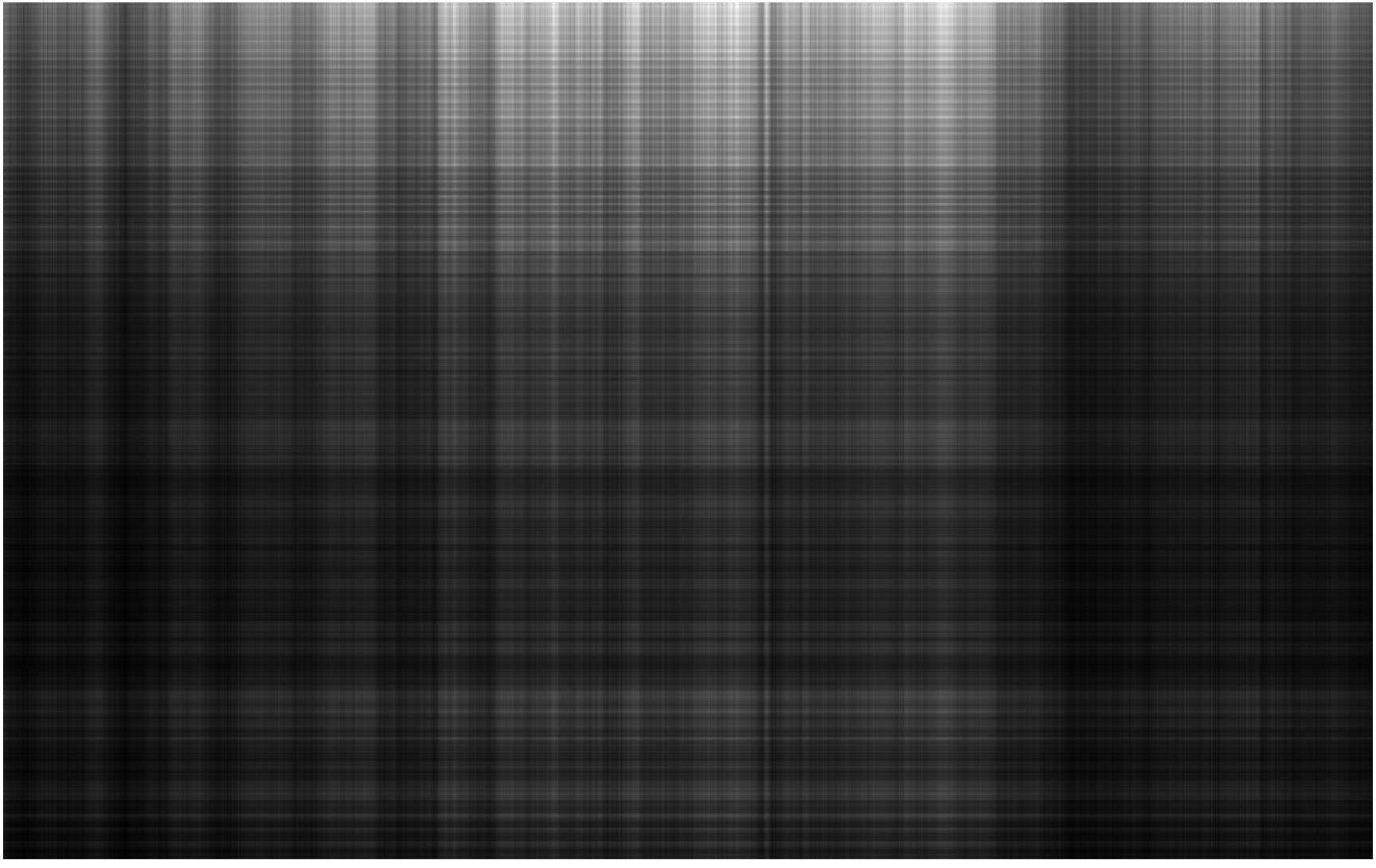
Collaborative Filtering, Matrix Completion



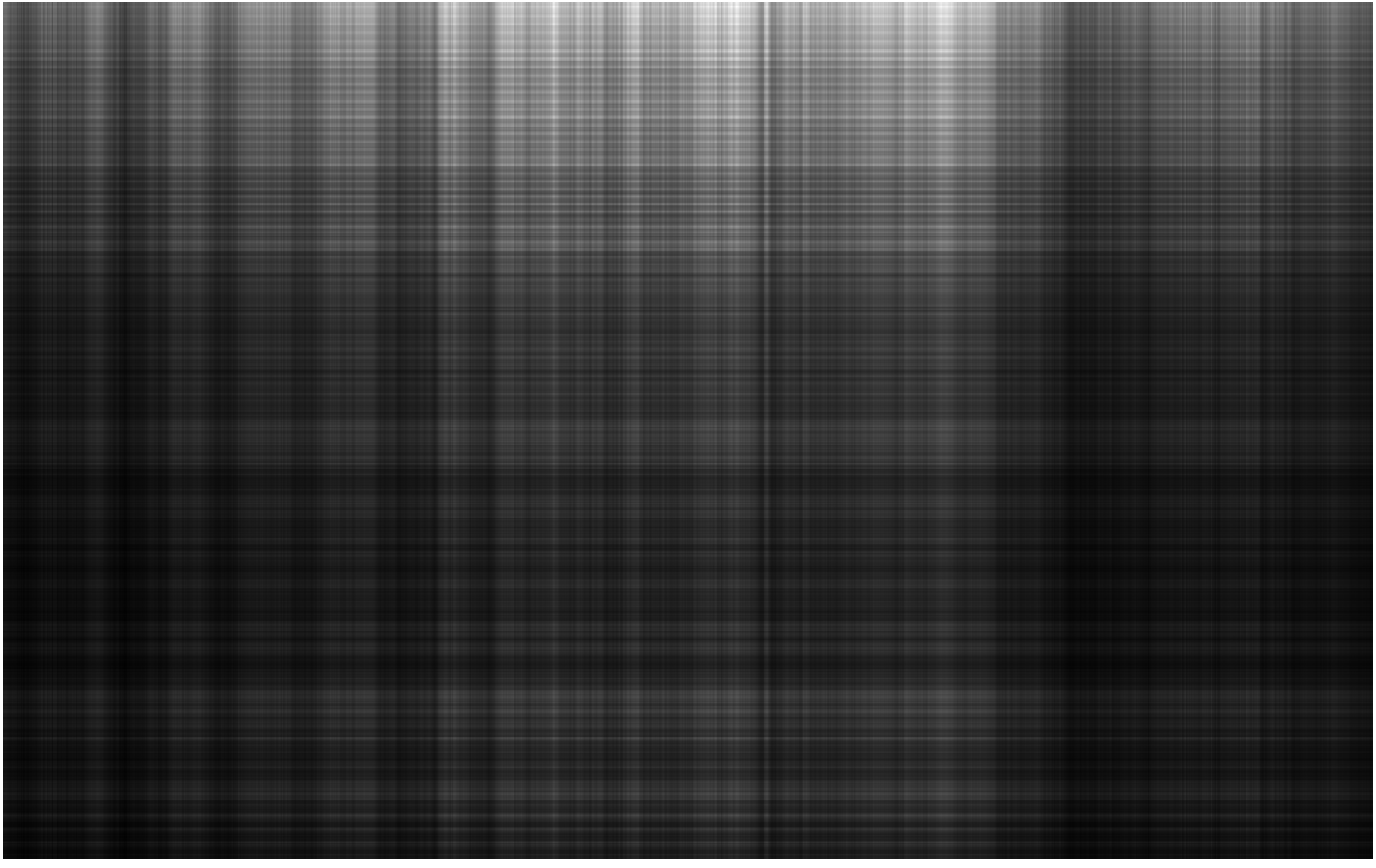
Collaborative Filtering, Matrix Completion



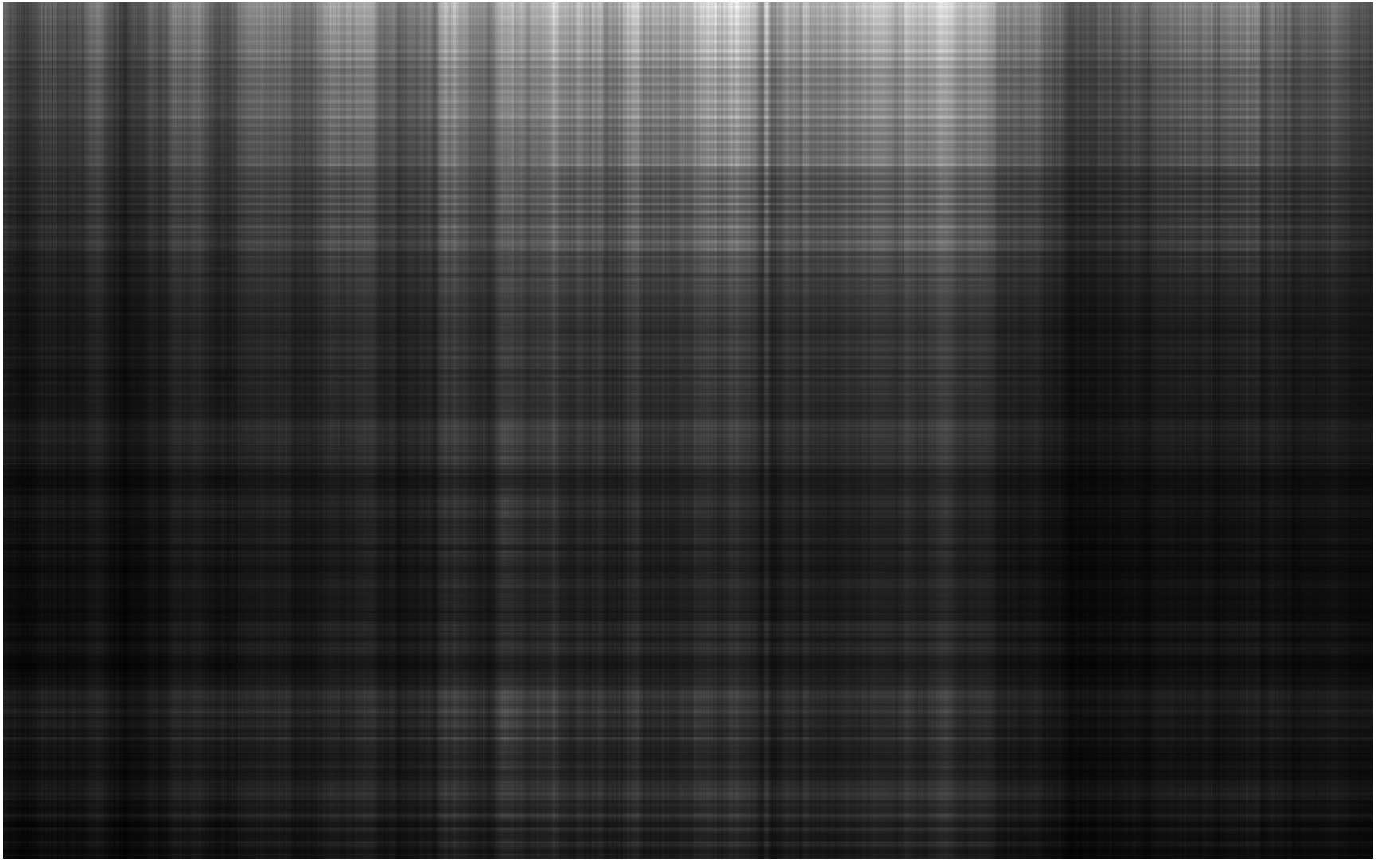
Collaborative Filtering, Matrix Completion



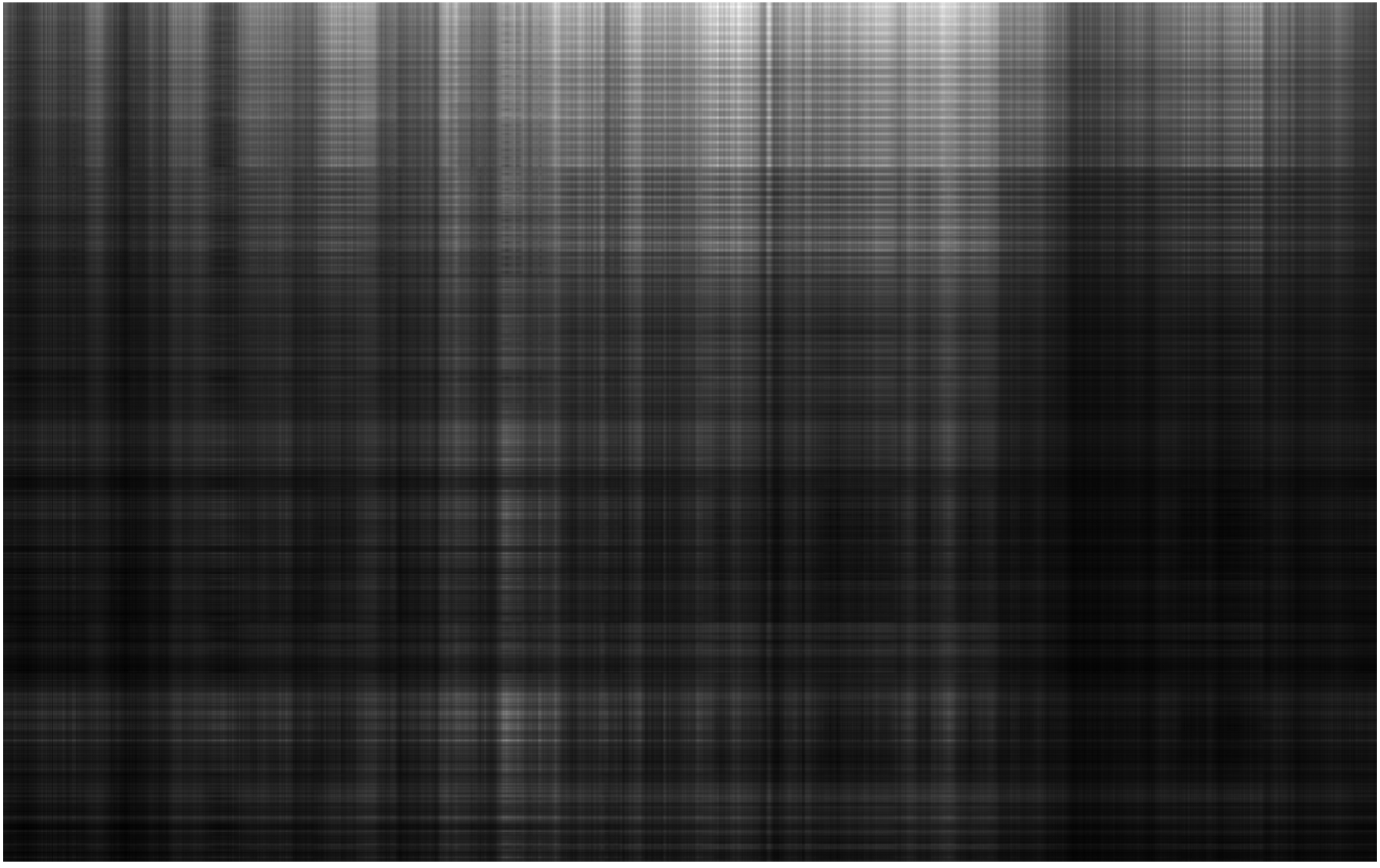
Collaborative Filtering, Matrix Completion



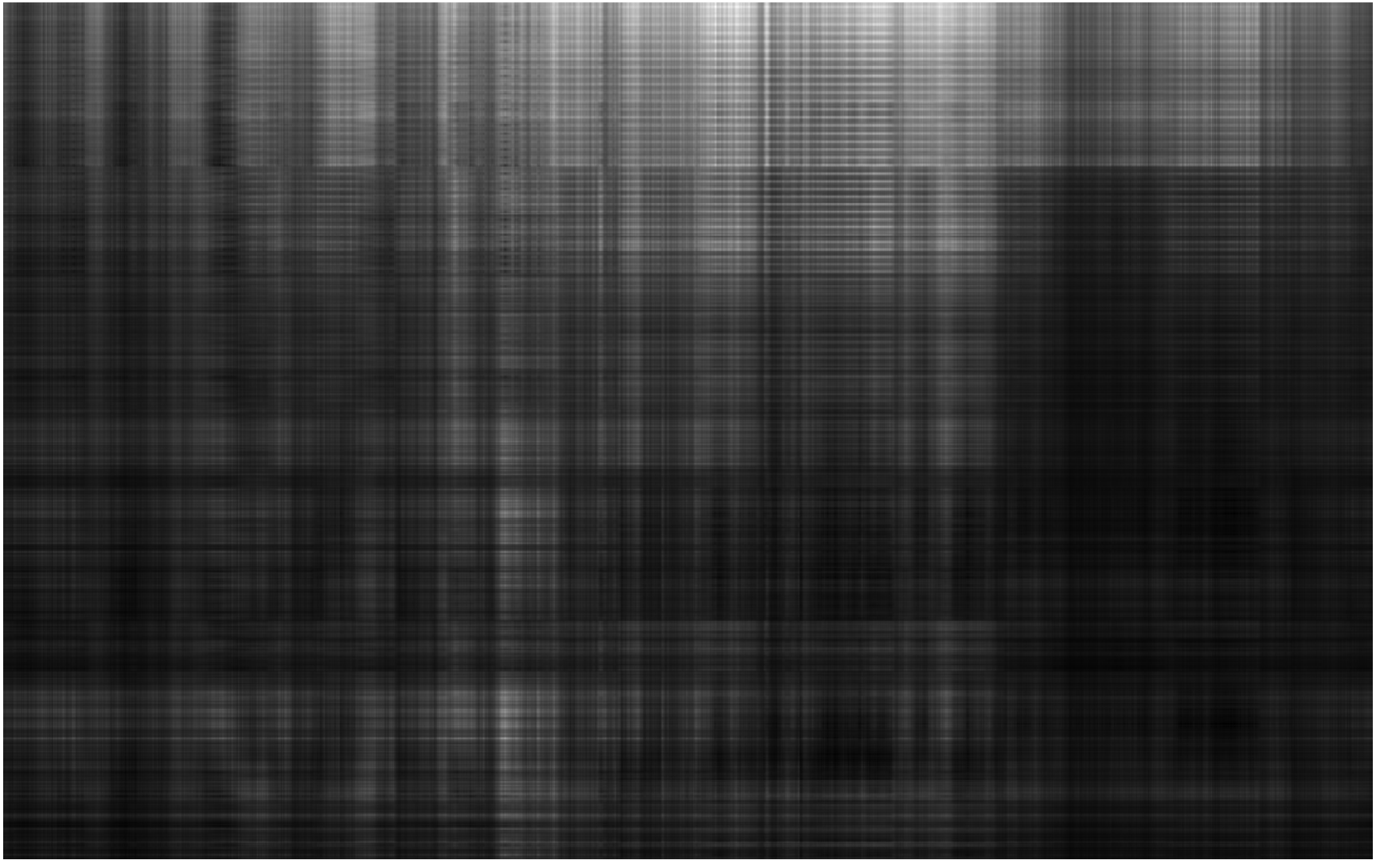
Collaborative Filtering, Matrix Completion



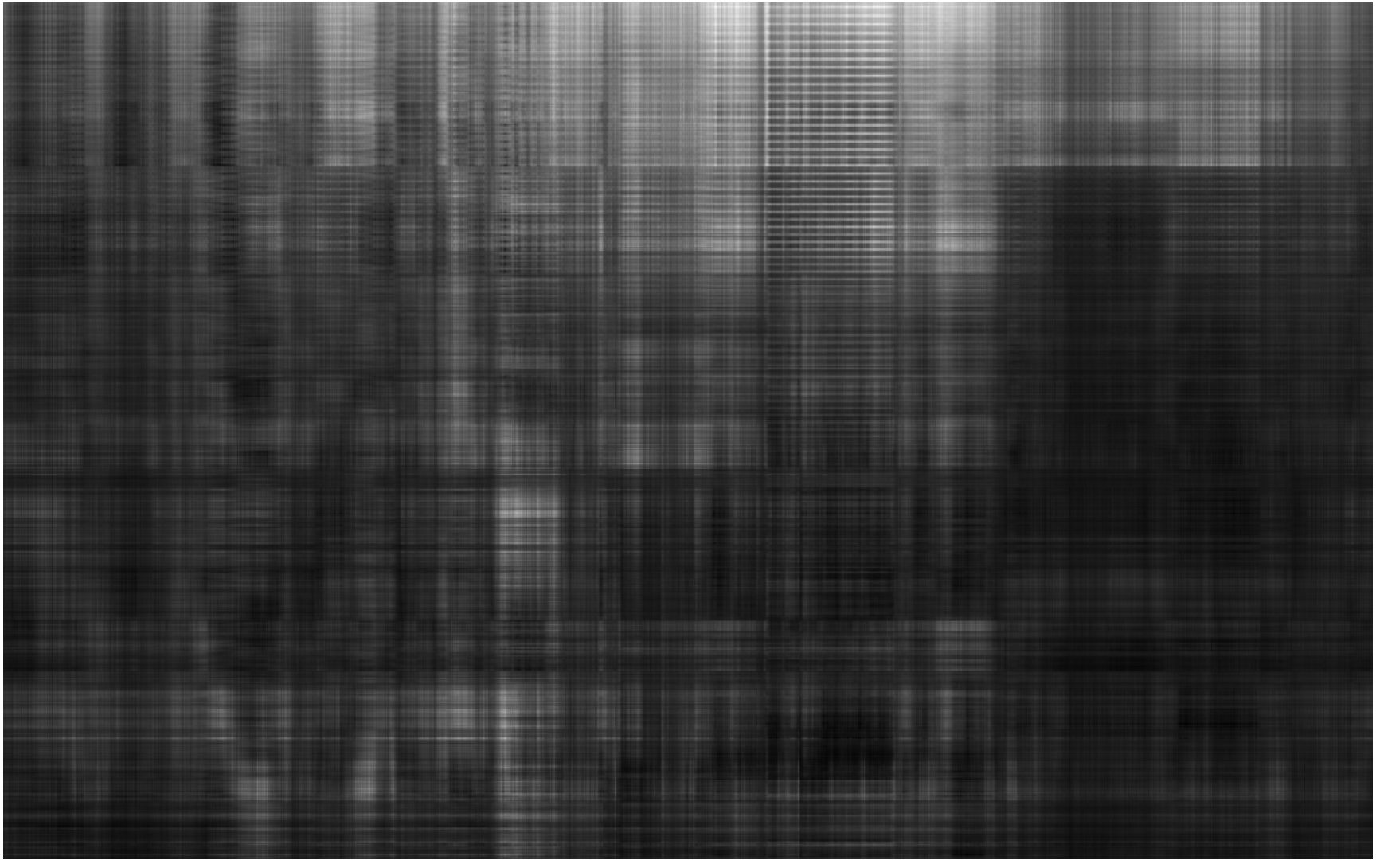
Collaborative Filtering, Matrix Completion



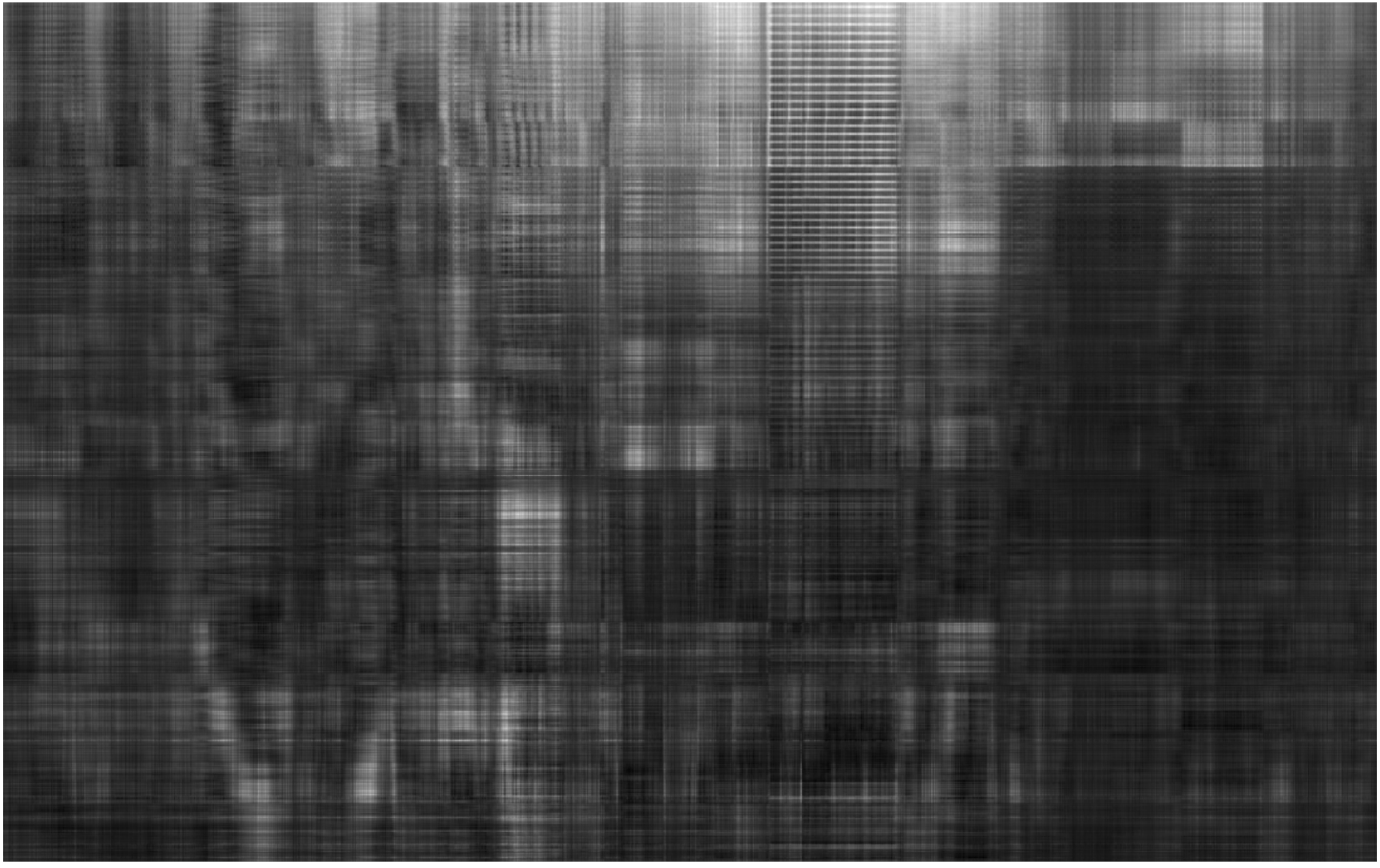
Collaborative Filtering, Matrix Completion



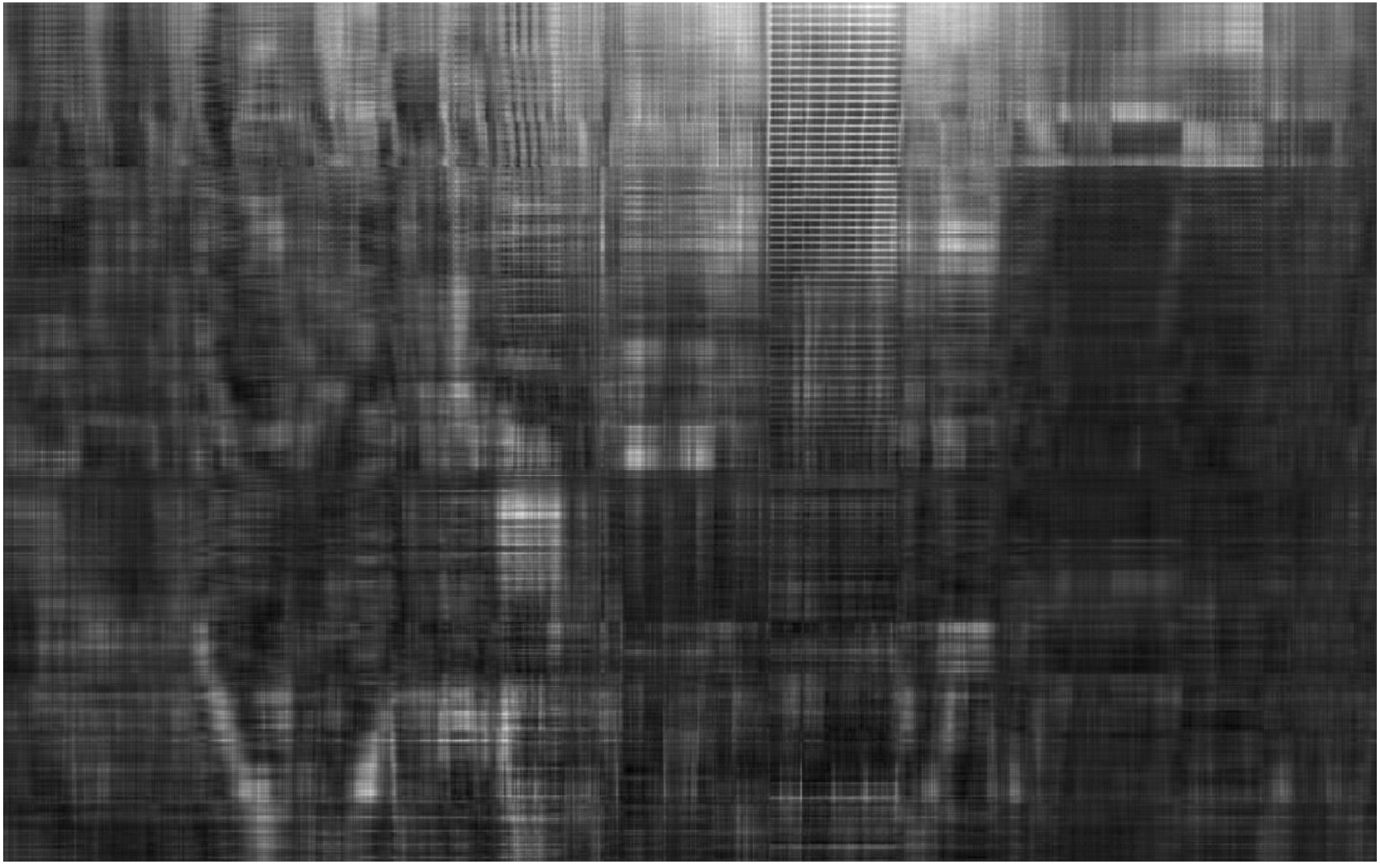
Collaborative Filtering, Matrix Completion



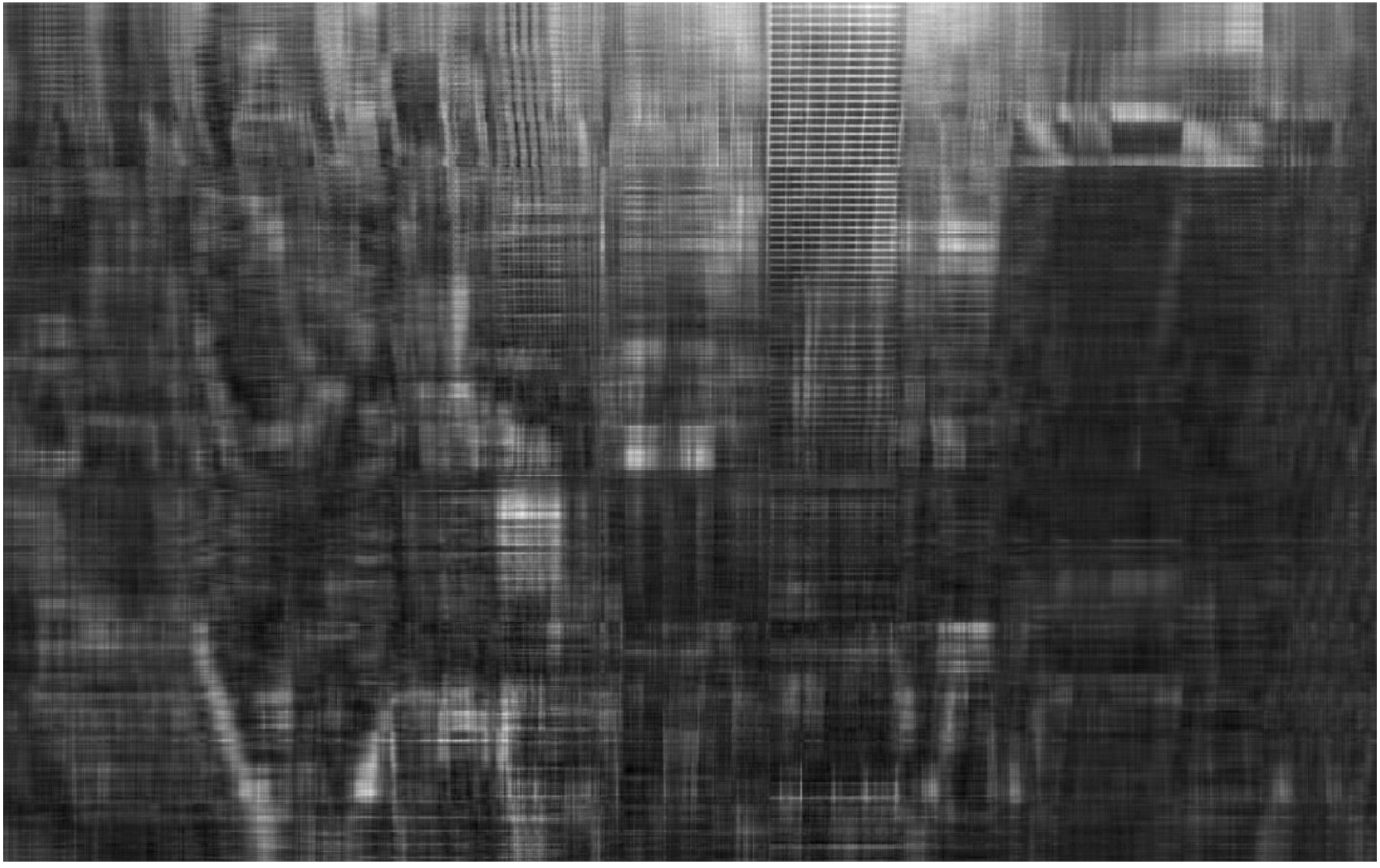
Collaborative Filtering, Matrix Completion



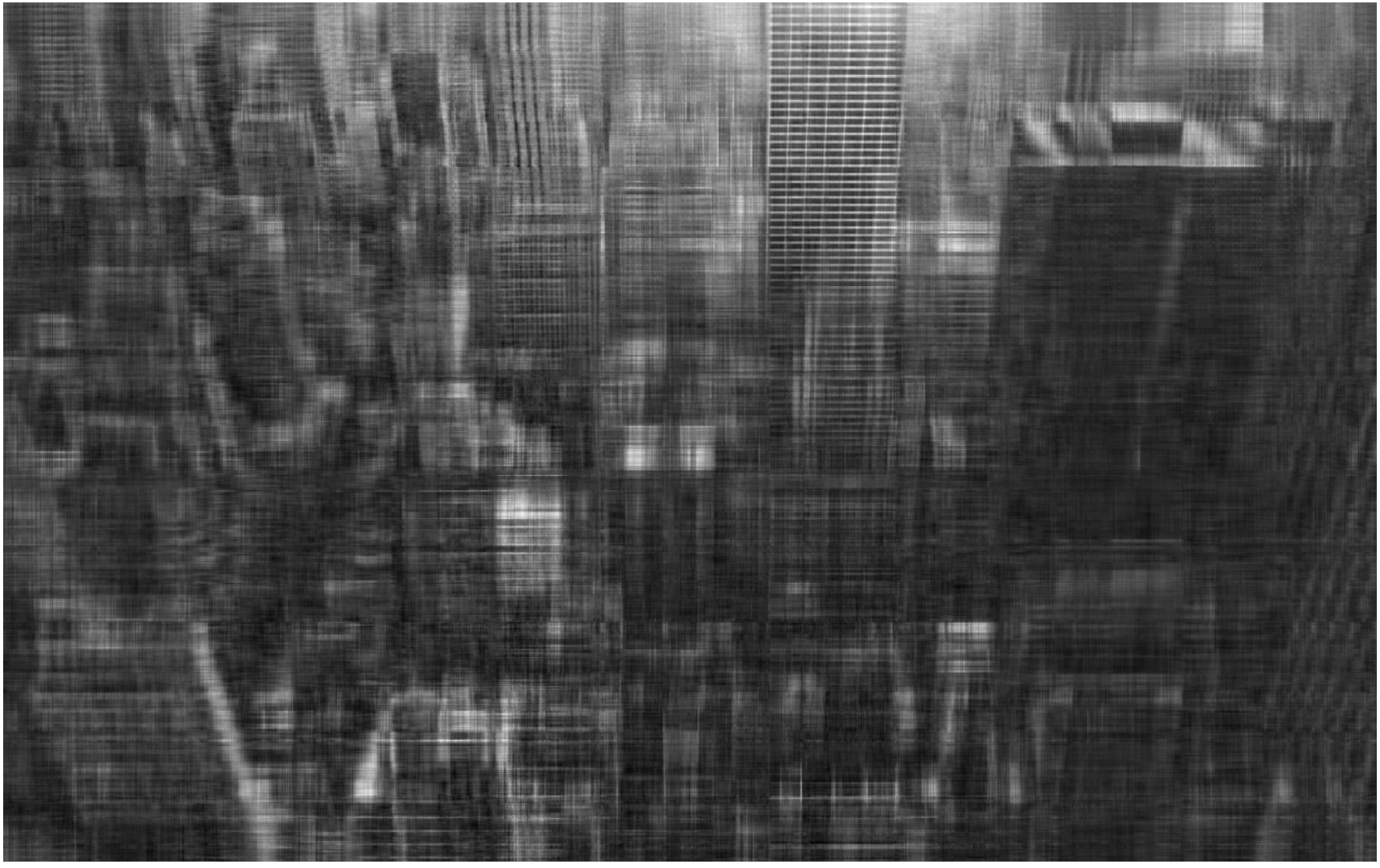
Collaborative Filtering, Matrix Completion



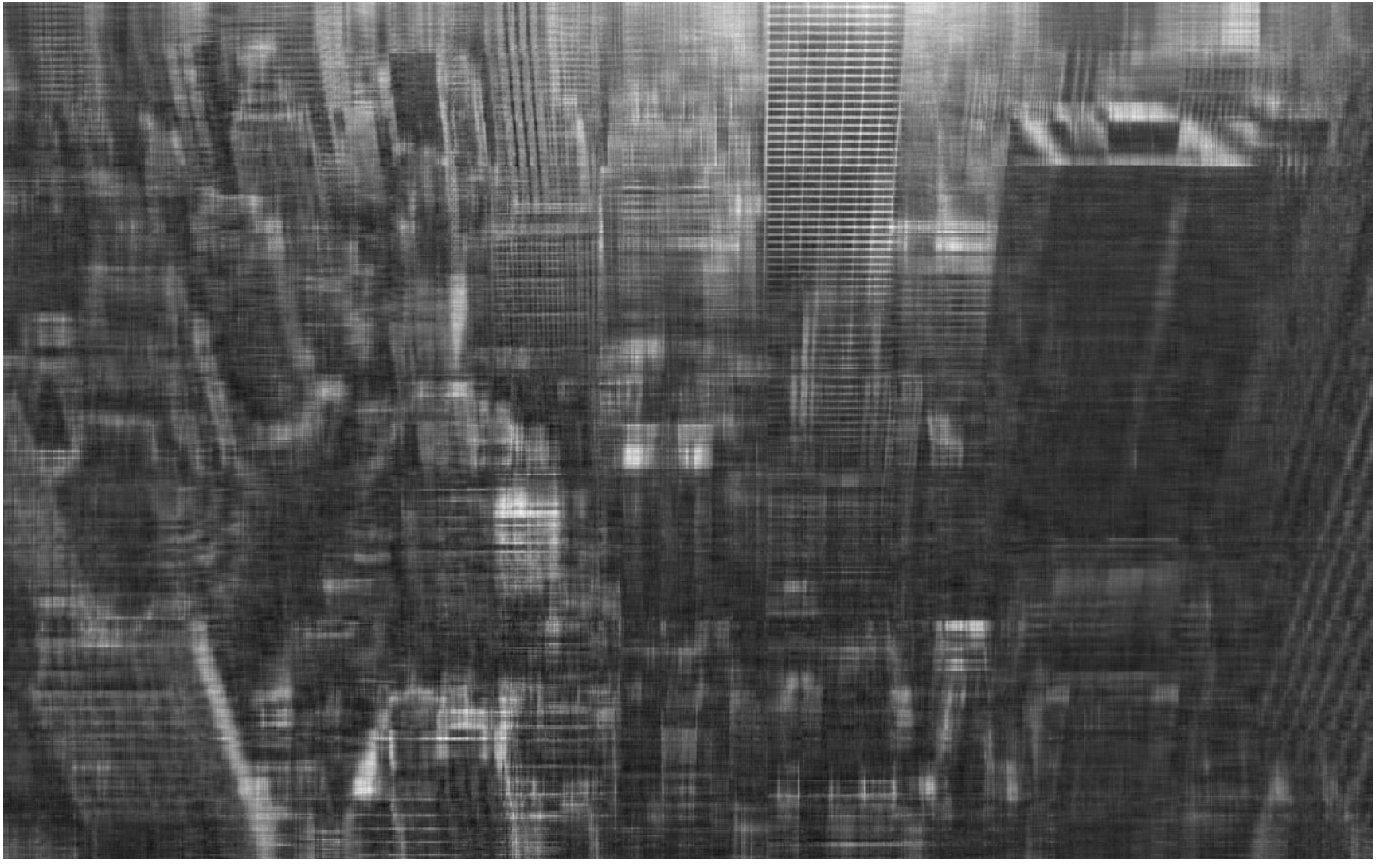
Collaborative Filtering, Matrix Completion



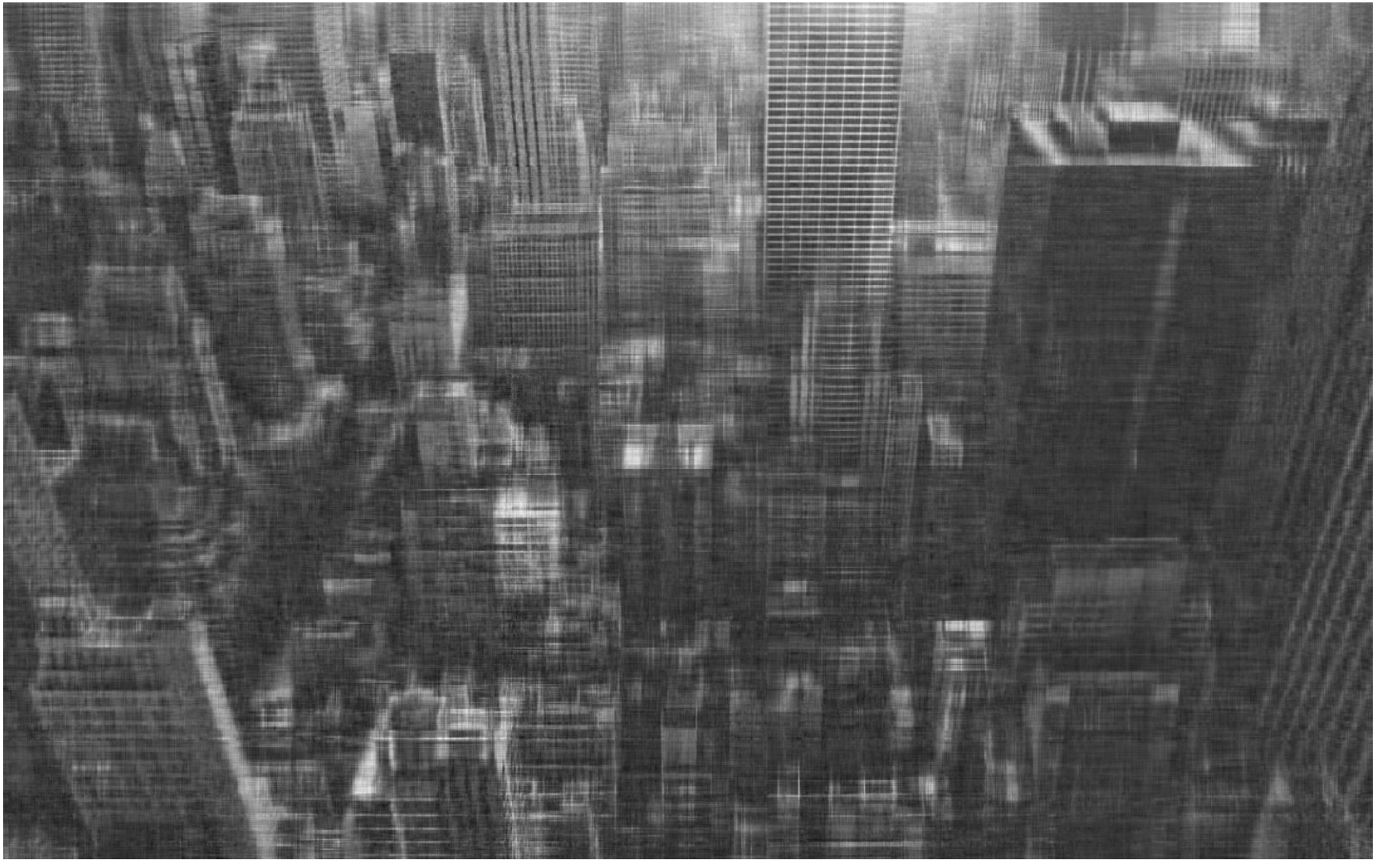
Collaborative Filtering, Matrix Completion



Collaborative Filtering, Matrix Completion



Collaborative Filtering, Matrix Completion



Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Collaborative Filtering, Matrix Completion

Exact reconstruction (no noise)

$$\hat{R} \in \operatorname{argmin}\{\|M\|_* \text{ such that } \mathcal{P}_E(M) = \mathcal{P}_E(R)\}$$

Assume $n = n_U = n_I$ for short and put $m = |E|$. Then under some assumptions

- No method can suceed if $m \leq crn \log n$. Namely, need **at least**

  $$m \geq crn \log n$$

  observed entries to recover M and $r = \text{rank}$
- If $m \geq crn(\log n)^2$ then reconstruction is **exact**! with a large probability
- In this setting, convex relaxation is **exact**: **no loss** when relaxing rank by $\|\cdot\|_*$
- Gives the exact same solution as the one constrained by rank!
- Convex programming incredibly powerful in this case
- Compressed sensing theory

Thank you!