Recommendation systems via approximate matrix factorization

Group 1 Report for HDDA F24 course

Dmitry Beresnev
AIDS-MS1, Innopolis University
`d.beresnev@innopolis.university`

Vsevolod Klyushev
AIDS-MS1, Innopolis University
`v.klyushev@innopolis.university`

1 Introduction

1.1 Problem formulation

We were given the following problem:

$$\min_{U \in \mathbb{R}^{m \times r}, V \in \mathbb{R}^{r \times n}} \|W \circ (X - UV)\|_F^2 \tag{1}$$

where $X \in \mathbb{R}^{m \times n}$ — target matrix, $W \in \mathbb{R}^{m \times n}$ — binary matrix (0 or 1), $r \in \mathbb{R}_+$ — rank of factorization.

1.2 Initialization of U, V

In order to have a good starting point, we decided to do the following:

- Impute unknown cells in matrix with mean among movies (via sklearn SimpleImputer)
- Perform randomized SVD decomposition (via sklearn randomized SVD) with required rank
- Take necessary number of first and last components of decomposition as U and V matrices correspondingly

2 Notations

Notation	Meaning
$\langle \cdot, \cdot \rangle$	dot product (Frobenius inner product in case of matrices)
$\| \cdot \|$	second norm (Frobenius norm in case of matrices)
$f, f(\cdot)$	objective function
$\nabla f, \nabla f(\cdot)$	gradient of objective function
X_i	i-th column of matrix X
$X_i^\top$	i-th row of matrix X

3 Gradient Descent

3.1 Gradients derivation

In order to use Gradient Descent method Algorithm 1, we need to compute gradients with respect to each parameter U and V. Using matrix-vector differentiation rules and some help from [2], we obtained the following:

$$\begin{aligned}\frac{\partial \|W \circ (X - UV)\|_F^2}{\partial U} &= -2(W \circ X)V^T + 2(W \circ (UV))V^T \\ \frac{\partial \|W \circ (X - UV)\|_F^2}{\partial V} &= -2U^T(W \circ X) + 2U^T(W \circ (UV))\end{aligned} \tag{2}$$

Algorithm 1 Gradient Descent optimizer

```
Input: $\theta_0$ (parameters to optimize), $f(\theta)$ (objective function), $\mathcal{L}(p)$ (step size choosing strategy)
  for $t = 1$ to ... do
    $p_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$                ▷ Step direction
    Choose step size $\gamma$ according to $\mathcal{L}(p_t)$
    $\theta_t \leftarrow \theta_{t-1} + \gamma p_t$
  end for
  return $\theta_t$
```

We have also added the regularization, so the final problem is

$$\min_{U \in \mathbb{R}^{m \times r}, V \in \mathbb{R}^{r \times n}} \|W \circ (X - UV)\|_F^2 + \lambda \|U\|_F^2 + \lambda \|V\|_F^2 \tag{3}$$

where λ — regularization parameter.

Finally, gradients of Equation (2) is computed as

$$\begin{aligned}\frac{\partial (\|W \circ (X - UV)\|_F^2 + \lambda \|U\|_F^2 + \lambda \|V\|_F^2)}{\partial U} &= -2(W \circ X)V^T + 2(W \circ (UV))V^T + 2\lambda U \\ \frac{\partial (\|W \circ (X - UV)\|_F^2 + \lambda \|U\|_F^2 + \lambda \|V\|_F^2)}{\partial V} &= -2U^T(W \circ X) + 2U^T(W \circ (UV)) + 2\lambda V\end{aligned} \tag{4}$$

The overall method is then the following: on each iteration we:

1. Fix V and update U using the first equation of Equation (4) and Algorithm 1
2. Fix U and update V using the second equation of Equation (4) and Algorithm 1

Since the objective function is $f : \mathbb{R}^{m \times n} \to \mathbb{R}$, the gradients are matrices. However, some step sizes strategies and optimizers uses dot product (for example, between descent direction and gradient). Therefore, hereafter, we substitute, where it is necessary, vector dot product with Frobenius inner product, defined as:

$$\langle A, B \rangle_F = Tr(A^T B) \tag{5}$$

3.2 Step size strategies

We decided to test different strategies for finding the decent step size α.

Constant step size

$$\alpha_k = \alpha$$

Decreasing step size

$$\alpha_k = \frac{1}{k}$$

or, alternatively,

$$\alpha_k = \frac{1}{\sqrt{k}}$$

Estimation of Lipschitz constant We know that for L-smooth function f, the following holds:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|, \forall x, y$$

Also, we know that in case of L-smooth function, the optimal step size would be $\alpha_k = \frac{1}{L}$. Hence, we can try to iteratively estimate it, what is done in Algorithm 3.

Armijo rule A step length α is said to satisfy the Armijo condition, restricted to the descent direction p ($p^T \nabla f(x) < 0$) if the following holds:

$$f(x + \alpha p) \leq f(x) + c_1 \alpha \langle p, \nabla f(x) \rangle \tag{6}$$

where $c_1 > 0$ is typically small (about 0.01) constant.

The Armijo rule ensures the 'sufficient' decrease in the function after making a step (Figure 1). Hence, Algorithm 4 iteratively decreases step size until it satisfies the Armijo condition.

Weak Wolfe conditions A step length α is said to satisfy the curvature condition, restricted to the descent direction p if the following holds:

$$-\langle p, \nabla f(x + \alpha) \rangle \leq -c_2 \langle p, \nabla f(x) \rangle \tag{7}$$

where $c_2 > 0$. The curvature condition helps to find points, in which our function is not decreasing as fast as in starting point Figure 2.

A step length α is said to satisfy the (Weak) Wolfe conditions (Figure 3) with $c_2 > c_1 > 0$, if both Equations (6) and (7) holds. Therefore, we can iteratively decrease step size until it satisfies the Weak Wolfe conditions. In [1] the efficient bisection algorithm (Algorithm 5) for such procedure is presented.

Strong Wolfe conditions However, the Wolfe conditions can result in a value for the step length that is not close to a minimizer of $\phi(\alpha) = f(x + \alpha p)$. We can modify the curvature condition as following:

$$\|\langle p, \nabla f(x + \alpha) \rangle\| \leq c_2 \|\langle p, \nabla f(x) \rangle\| \tag{8}$$

A step length α is said to satisfy the Strong Wolfe conditions (Figure 4) with $c_2 > c_1 > 0$, if both Equations (6) and (8) holds.These conditions force α to lie close to a critical point of ϕ. The algorithm that searches for points satisfying the String Wolfe Conditions is presented in Algorithm 6.

3.3 Optimizers

We have also decided to try other optimizer rather than only Gradient Descent. New optimization algorithms can help to improve the convergence and avoid local minima by introducing either the idea of momentum (Algorithms 8 and 9), or adaptivity (Algorithms 7 and 10 to 12).

Adaptive gradient descent Relatively new approach, proposed in 2020 in paper [4], which is more the step size search approach, than the true optimizer. This method has several advantages:

- It is not iterative
- It relies on local smoothness (as a method of estimating $\frac{1}{L}$)

. In a nutshell, the algorithm Algorithm 7 at each iteration k tries to find the maximum step size α_k satisfying

$$\begin{gathered} \frac{\alpha_k^2}{\alpha_{k-1}^2} \leq (1 + \theta_{k-1}) \\ \alpha_k \leq \frac{\|x^k - x^{k-1}\|}{2\|\nabla f(x^k) - \nabla f(x^{k-1})} \end{gathered} \tag{9}$$

where $\theta_k = \frac{\alpha_k}{\alpha_{k-1}}$.

The first inequality of Equation (9) makes sure new step size not to be too big compared to previous. The second inequality of Equation (9) sets upper bound for step to be $\frac{1}{2L}$, where L is that the smoothness condition holds for the current and the precious points.

Heavy Ball In theory is usually written as

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) + \beta_k (x_k - x_{k-1})$$

where β — momentum, which determines the effect of the previous steps on the current update.

The Heavy Ball method was proposed in 1964 by Boris Polyak. This is a classic improvement of the standard Gradient Descent by adding momentum: the method combines the current gradient with the previous history to accelerate convergence.

Realization is presented in Algorithm 8.

Nesterov momentum In theory is usually written as

$$\begin{aligned} x_{k+1} &= y_k - \alpha_k \nabla f(y_k) \\ y_{k+1} &= x_{k+1} + \beta_k (x_{k+1} - x_k) \end{aligned}$$

where β_k — momentum, usually taken constant $\beta_k = \beta$.

The Nesterov Accelerated Method extends the idea of momentum in Gradient Descent. It was introduced by Yuri Nesterov in 1983. The basic idea is to use partial prediction when updating parameters, including momentum to accelerate convergence. The key difference from the heavy ball method is that due to the impulse to the gradient counting point, extrapolation, or 'looking into the future' occurs.

Realization is presented in Algorithm 9.

AdaGrad

$$\begin{aligned} g_k &= \nabla f(x_k) \\ G_k &= G_{k-1} + g_k^2 \\ x_{k+1} &= x_k - \frac{\gamma_k}{\sqrt{G_k} + \epsilon} g_k \end{aligned}$$

where G_k — cumulative squared sum of gradients, ϵ — small constant for stability.

AdaGrad was developed to solve the problems associated with setting a uniform learning rate for all parameters, especially in situations where the features have significantly different frequency or importance: it would be good to be able to update the parameters with an eye to how typical a feature they capture.

Realization is presented in Algorithm 10.

RMSprop

$$\begin{aligned} g_k &= \nabla f(x_k) \\ G_k &= \beta G_{k-1} + (1-\beta) g_k^2 \\ x_{k+1} &= x_k - \frac{\gamma_k}{\sqrt{G_k} + \epsilon} g_k \end{aligned}$$

where G_k — weighted moving average of squared gradients.

The key idea of RMSProp is to scale the gradient of each weight in the model by dividing it by the root mean square value of the gradients of that weight. This helps prevent weights with high gradients from learning too quickly, while at the same time allowing weights with low gradients to continue learning faster. Method solves the main problem of AdaGrad — old gradients and recent gradients have different weights, that is, there is a kind of 'forgetting' of history.

Realization is presented in Algorithm 11.

Adam

$$\begin{aligned}
&g_k = \nabla f(x_k)\\
&m_k = \beta_1 m_{k-1} + (1-\beta_1) g_k\\
&v_k = \beta_2 v_{k-1} + (1-\beta_2) g_k^2\\
&\hat{m_k} = \frac{m_k}{1-\beta_1^k}\\
&\hat{v_k} = \frac{v_k}{1-\beta_2^k}\\
&x_{k+1} = x_k - \frac{\gamma_k}{\sqrt{\hat{v_k}}+\epsilon}\hat{m_k}
\end{aligned}$$

where m_k, v_k — so called first and second momentum of gradient.

Adam supports exponential moving averages of weights and gradients, which it uses to scale the learning rate: it uses estimates of the mean and variance of gradients to adaptively scale the learning rate during training.

If we consider the following type of formula for correction ($\hat{v_k}$):

$$\hat{v_k} = \frac{(1-\beta_2)\sum_{i=1}^{k}\beta_2^{k-i}\text{diag}(g_i \odot g_i)}{1-\beta_2^k}.$$

one can notice that g_k^2 is taken with a weight of 1, g_{k-1}^2 is taken with a weight of β_2, g_{k-2}^2 is taken with a weight of β_2^2, and so on. Next, all weights are divided by $1+\beta_2+\beta_2^2+\cdots+\beta_2^k = \frac{1-\beta_2^k}{(1-\beta_2)}$. Thus, the sum of all the weights is equal to 1, that is, a convex combination is obtained. Therefore, the weights for the gradients depend on the iteration number. In the initial iterations, it behaves in a similar way to AdaGrad, and in the later iterations it becomes similar to RMSprop.

Realization is presented in Algorithm 12.

BFGS The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm is a quasi-Newton method, that approximates the Hessian B_k of the objective function. The authors suggest to compute $H_k = B_k^{-1}$ by solving

$$\begin{aligned}
&\min_H \|H_k - H\|\\
\text{s.t. } &H = H^\top\\
&Hy_k = s_k
\end{aligned} \tag{10}$$

where $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ and $s_k = x_{k+1} - x_k$.

The suggestion is quite intuitive: let us look for the H_k that is closest in any norm to the real inverse Hessian, satisfies the quasi-Newtonian equation (last equality in Equation (10)) and is symmetric (like the real inverted Hessian itself).

The authors proved that a two-rank Hessian update is enough:

$$B_{k+1} = B_k + \alpha uu^T + \beta vv^T$$

And one can obtain the equation for the inverted hessian:

$$H_{k+1} = (I - \rho_k s_k y_k^\top) H_k (I - \rho_k y_k s_k^\top) + \rho_k s_k s_k^\top,$$

where $\rho_k = \dfrac{1}{y_k^\top s_k}$.

Realization is presented in Algorithm 13.

Note, that we can not directly use BFGS for the methods Equations (1) and (3), because the approximated hessian (and its inverse) would be tensors as second derivatives of function $f : \mathbb{R}^{m\times n} \to \mathbb{R}$. However, this methods can be used for Section 4.

4 Vector Gradient Descent

Instead of making update for entire U or V simultaneously, we can make updates row by row (column by column for V). The reasons are the following:

1. The objective function becomes $f : \mathbb{R}^d \to \mathbb{R}$, so we can apply methods like BFGS (Algorithm 13)
2. There will be more updates, and such updates will be more diverse: we will use just updated values for new updates

Therefore, the new problem with fixed V becomes

$$\min_{U_i^\top \in \mathbb{R}^r} \|W_i^\top \circ (X_i^\top - U_i^\top V)\|^2 + \lambda \|U_i^\top\|^2,\ \forall i \in \{1, 2, \dots, m\} \tag{11}$$

and the new problem with fixed U:

$$\min_{V_j \in \mathbb{R}^r} \|W_j \circ (X_j - UV_j)\|^2 + \lambda \|V_j\|^2,\ \forall j \in \{1, 2, \dots, n\} \tag{12}$$

Hence gradient of Equation (11) is

$$\frac{\partial(\|W_i^\top \circ (X_i^\top - U_i^\top V)\|^2 + \lambda \|U_i^\top\|^2)}{\partial U_i^\top} = -2(W_i^\top \circ X_i^\top)V^T + 2(W_i^\top \circ (U_i^\top V))V^T + 2\lambda U_i^\top \tag{13}$$

and the gradient of Equation (12):

$$\frac{\partial(\|W_j \circ (X_j - UV_j)\|^2 + \lambda \|V_j\|^2)}{\partial V_j} = -2U^T(W_j \circ X_j) + 2U^T(W_j \circ (UV_j)) + 2\lambda V_j \tag{14}$$

Obviously, if we want to obtain more diverse updates, we should change the order of U and V updates: if we firstly update all rows of U and only then all the columns of V, the updates would be the same as in Equation (4).

Remember, $X \in \mathbb{R}^{m \times n}$, what means that number of rows of U and number of columns of V has the ratio $\frac{m}{n} \approx \frac{k_U}{k_V}$, where k_U and k_V are possibly small integers. Therefore,it is reasonable to firstly update k_U rows of U and then update k_V columns of V. For the given problem, $X \in \mathbb{R}^{6040 \times 3952}$, so $k_U \approx 3$ and $k_V \approx 2$ The final algorithm, what we called Vector Gradient Descent, then looks as presented in Algorithm 2.

Algorithm 2 Vector Gradient Descent

```
Input: $X, W \in \mathbb{R}^{m \times n}$ — given initial and binary matrices, $U \in \mathbb{R}^{m \times r}, V \in \mathbb{R}^{r \times n}$ — arbitrary matrices, $k_U$, $k_V$ — small integers such that $\frac{m}{n} \approx \frac{k_U}{k_V}$

  $d \leftarrow k_U + k_V$
  repeat
      for $t = 0$ to $n + m - 1$ do
          $r \leftarrow t$ mod $d$
          if $r < k_U$ then
              $i \leftarrow k_U \cdot (t$ div $d) + r + 1$
              if $i > m$ then
                  continue
              end if
              Update $U_i^\top$ using Equation (13)
          else
              $j \leftarrow k_V \cdot (t$ div $d) + r - k_U + 1$
              if $j > n$ then
                  continue
              end if
              Update $V_j$ using Equation (14)
          end if
      end for
  until convergence

  return $U, V$
```

5 Non-Negative Matrix Factorization (NNMF)

Let us add to the initial problem (Equation (1)) non-negativity constraints:

$$\begin{aligned} &\min_{U\in R^{m\times r}, V\in R^{r\times n}} \|W\circ(X-UV)\|_F^2 \\ &\text{s.t. } U,V\geq 0 \end{aligned} \tag{15}$$

In order to solve such problem more easily, we need to derive multiplicative updates for U and V.

Firstly, let us recall gradient update from the previous parts:

$$U \leftarrow U + \alpha_u((W\circ X)V^T - (W\circ(UV))V^T) \tag{16}$$

If we set $\alpha_u = \dfrac{U}{((W\circ(UV))V^T)}$, we get:

$$\begin{aligned} U &\leftarrow U + \alpha_u((W\circ X)V^T - (W\circ(UV))V^T) = \\ &= U + U\circ\frac{((W\circ X)V^T)}{((W\circ(UV))V^T)} - U\circ\frac{((W\circ(UV))V^T)}{((W\circ(UV))V^T)} = \\ &= U + U\circ\frac{((W\circ X)V^T)}{((W\circ(UV))V^T)} - U = \\ &= U\circ\frac{((W\circ X)V^T)}{((W\circ(UV))V^T)} \end{aligned} \tag{17}$$

We have achieved multiplicative update for U. Now, let us do the same for V:

$$V \leftarrow V + \alpha_v(U^T(W\circ X) - U^T(W\circ(UV))) \tag{18}$$

If we'll use $\alpha_v = \dfrac{V}{(U^T(W\circ(UV)))}$, then we'll get:

$$\begin{aligned} V &\leftarrow V + \alpha_v(U^T(W\circ X) - U^T(W\circ(UV))) = \\ &= V + V\circ\frac{(U^T(W\circ X))}{(U^T(W\circ(UV)))} - V\circ\frac{(U^T(W\circ(UV)))}{(U^T(W\circ(UV)))} = \\ &= V + V\circ\frac{(U^T(W\circ X))}{(U^T(W\circ(UV)))} - V = \\ &= V\circ\frac{(U^T(W\circ X))}{(U^T(W\circ(UV)))} \end{aligned} \tag{19}$$

Now we also have achieved multiplicative update for V.

Such idea was used in Lee and Seung paper [3] for problem without mask in order to avoid subtraction, so the production of negative elements. Authors provided the proof of convergence for such adaptive learning rates for problems without mask.

However, our binary mask W brings some problems with division operation (the division on zero or very small number is possible). Adding some small constant ϵ to denominator captures this. This brings our multiplicative updates to be in the following forms:

$$U \leftarrow U\circ\frac{((W\circ X)V^T)}{((W\circ(UV))V^T+\epsilon)} \tag{20}$$

$$V \leftarrow V\circ\frac{(U^T(W\circ X))}{(U^T(W\circ(UV))+\epsilon)} \tag{21}$$

However, results demonstrated by such method is not satisfactory: it over-fits on the first iteration and shows very bad performance on test set (Figure 14).

6 Neural Network

As an alternative approach we decided to train simple neural network with the following parameters:

- 4 layers $(23 \times 64, 64 \times 128, 128 \times 64, 64 \times 1)$
- ReLU as activation function after each layer
- MSELoss criterion
- Adam optimizer with step size $\alpha = 0.001$
- Early stopping (if on test set loss is not decreasing for 3 iterations)
- Maximum number of training epochs $k = 50$

Model input consists of:

- Min-max scaled ‘user_id‘
- Min-max scaled ‘movie_id‘
- One-hot encoded ‘genres‘ (we have 18 unique genres)
- Binary encoded ‘gender‘
- Min-max scaled first two user features (‘feature_1‘ and ‘feature_2‘)

As output model returns just one number — rating for specific pair of user and movie.

Model was able to achieve average MSE loss on test set ≈ 1.1

7 Results

We have tested several approaches: Gradient descent (Section 3), Vector Gradient descent (Section 4), Non-Negative Matrix Factorization (Section 5) and Neural Network (section 6).

Both Neural Network and Vector Gradient descent showed their applicability for recommendation system task, however, they have been outperformed by other models.

As for NNMF, after several experiments we came to the conclusion, that such approach is inapplicable for recommendation system task mostly because of the mask in the problem formulation. Mask makes NNMF to set almost all unknown values to 1, which is quite a bad decision.

The Gradient descent method showed the best overall performance (both in terms of time and RMSE score on test data). Among all different optimizers, step selection strategies and other hyper-parameters (you can check experiments in Appendix D) we found the best composition: Gradient descent method for $r = 10$ with estimate 1/L strategy and $\lambda = 2$. This method achieved RMSE score 0.86 on test data.

References

[1] Anton Evgrafov. Convergence of descent methods with backtracking (armijo) linesearch. bisection algorithm for weak wolfe conditions.

[2] Kwan. Derivative of the frobenius norm of a matrix involving the hadamard products, 2020.

[3] Daniel Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000.

[4] Yura Malitsky and Konstantin Mishchenko. Adaptive gradient descent without descent. 2019.

Supplementary materials

A Step Size Conditions Figures

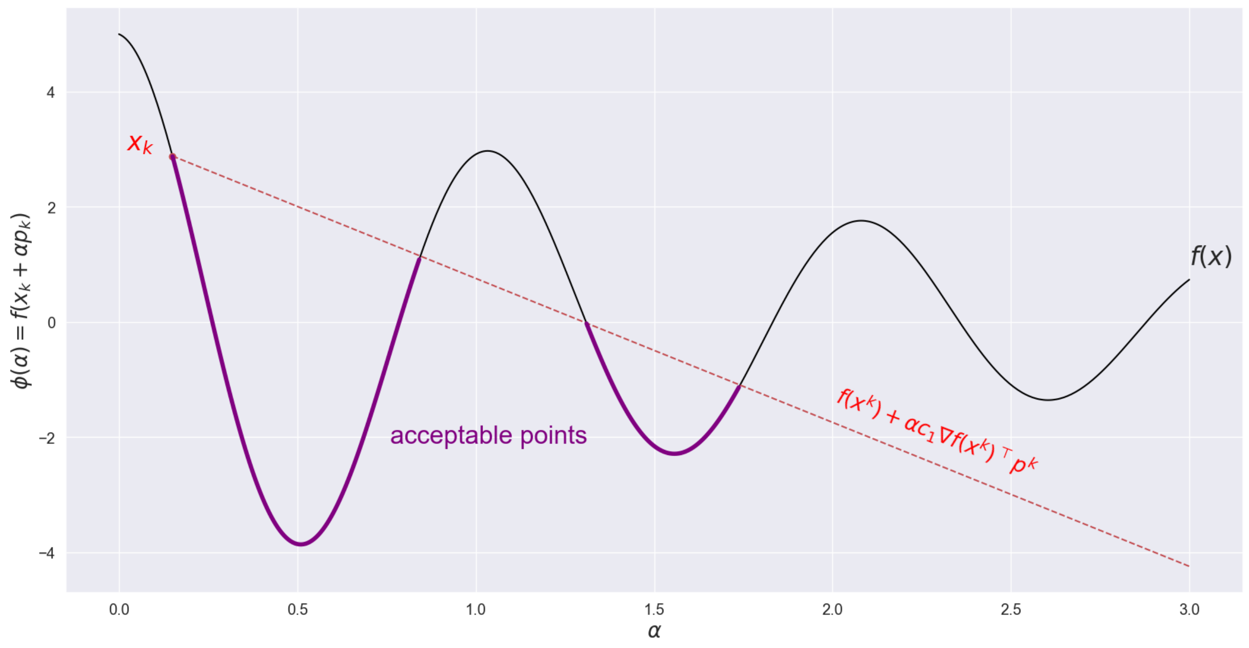

Figure 1: Armijo rule

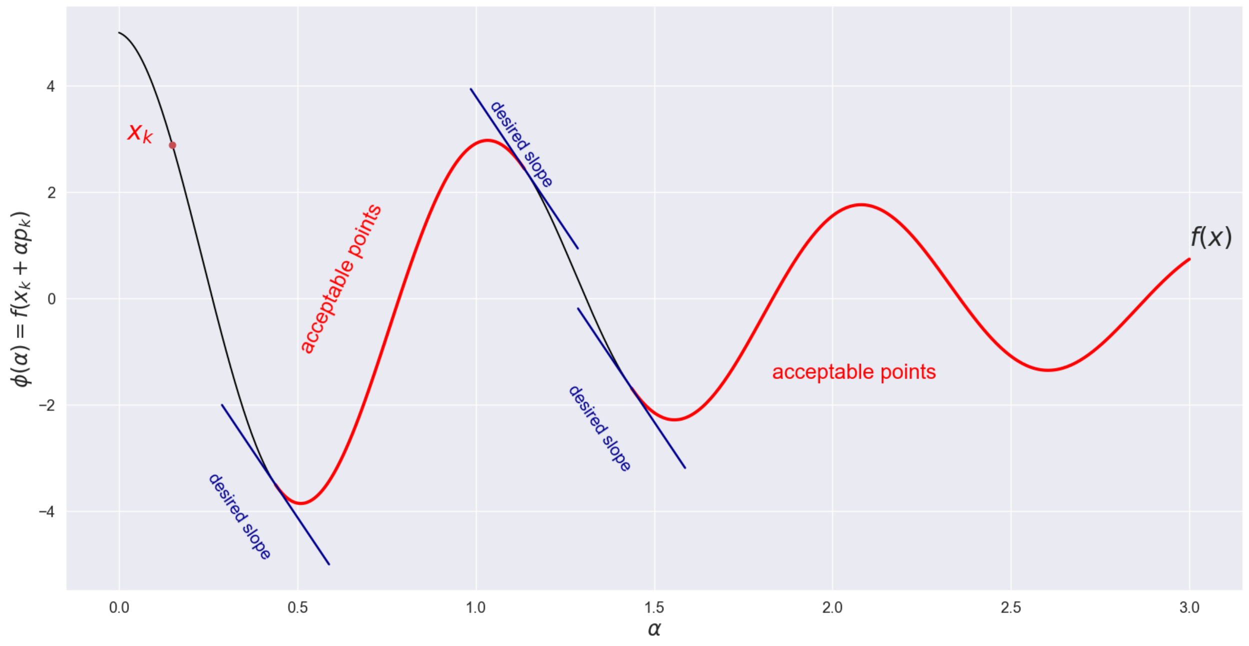

Figure 2: Curvature condition

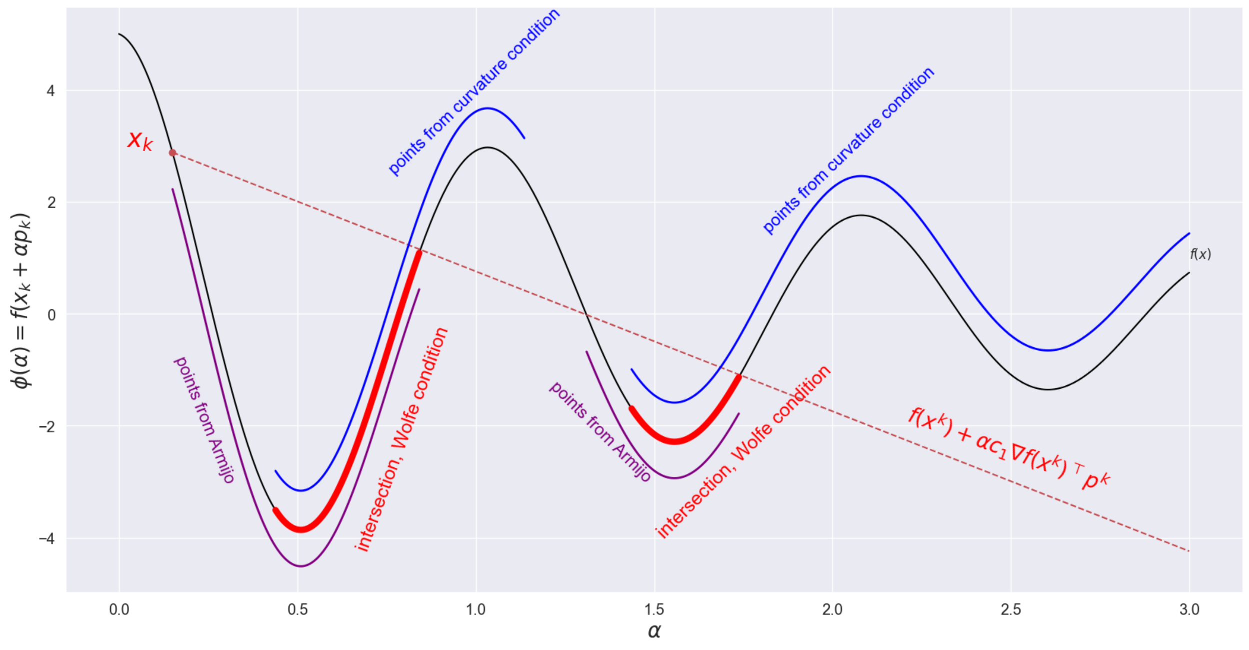

Figure 3: Weak Wolfe conditions

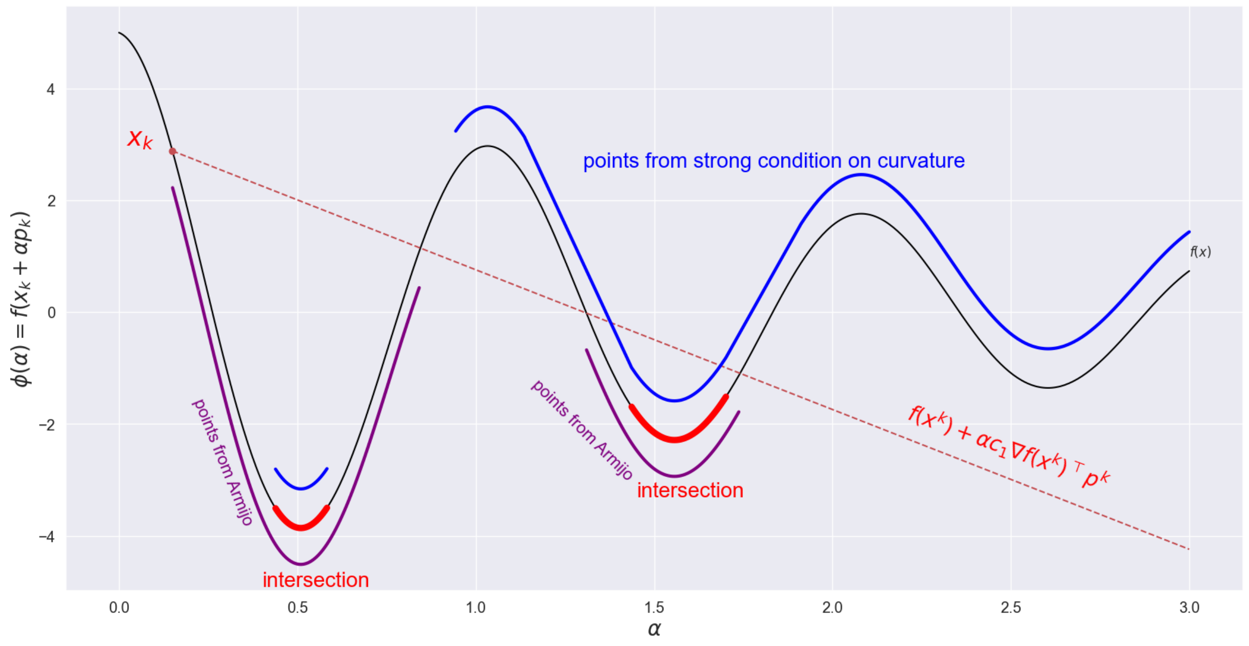

Figure 4: Strong Wolfe conditions

B Step selection Algorithms

Algorithm 3 Estimate $1/L$

```
Input: $\theta$ (point), $\nabla f(\theta)$ (gradient function), $p$ (desired direction), $\beta$ (multiplier), $t$ (max iterations)
  $\alpha \leftarrow 1$
  for $i = 1$ to $t$ do
      $\theta_i \leftarrow \theta + \alpha p$
      if $\|\nabla f(\theta) - \nabla f(\theta_i)\| > \frac{1}{\alpha}\|\theta - \theta_i\|$ then
          $\alpha \leftarrow \beta \cdot \alpha$
          if $\alpha < \epsilon$ then
              return $\alpha$
          end if
      else
          return $\alpha$
      end if
  end for
  return $\alpha$
```

Algorithm 4 Armijo Step

```
Input: $\theta$ (point), $f(\theta)$ (objective function), $p$ (desired direction), $\beta$ (multiplier), $c_1 > 0$, $t$ (max iterations)
  $\alpha \leftarrow 1$
  for $i = 1$ to $t$ do
      $\theta_i \leftarrow \theta + \alpha p$
      if $f(\theta_i) > f(\theta) + c_1 \alpha \langle \nabla_\theta f(\theta), p \rangle$ then
          $\alpha \leftarrow \beta \cdot \alpha$
          if $\alpha < \epsilon$ then
              return $\alpha$
          end if
      else
          return $\alpha$
      end if
  end for
  return $\alpha$
```

Algorithm 5 Bisection Weak Wolfe Step

```
Input: $\theta$ (point), $f(\theta)$ (objective function), $p$ (desired direction), $\beta$ (multiplier), $c_1 > 0$, $c_2 > c_1$, $t$ (max iterations)
  $\alpha \leftarrow 1$
  $a \leftarrow 0$                                                    ▷ Lower bound
  $b \leftarrow +\infty$                                              ▷ Upper bound
  for $i = 1$ to $t$ do
      $\theta_i \leftarrow \theta + \alpha p$
      if $f(\theta_i) > f(\theta) + c_1\alpha\langle\nabla_\theta f(\theta), p\rangle$ then    ▷ Armijo condition
          $b \leftarrow \alpha$
          $\alpha \leftarrow \frac{1}{2}(a + b)$                      ▷ Decrease step
      else if $\langle p, \nabla_\theta f(\theta_i)\rangle < c_2\langle p, \nabla_\theta f(\theta)\rangle$ then    ▷ Curvature condition
          $a \leftarrow \alpha$
          if $b = +\infty$ then
              $\alpha \leftarrow 2a$
          else
              $\alpha \leftarrow \frac{1}{2}(a + b)$                  ▷ Increase step
          end if
      else
          return $\alpha$
      end if
  end for
  return $\alpha$
```

Algorithm 6 Strong Wolfe Step

```
Input: $\theta$ (point), $f(\theta)$ (objective function), $p$ (desired direction), $\beta$ (multiplier), $c_1 > 0$, $c_2 > c_1$, $t$ (max iterations)
  $\alpha \leftarrow 1$
  for $i = 1$ to $t$ do
      $\theta_i \leftarrow \theta + \alpha p$
      if $\big(f(\theta_i) > f(\theta) + c_1\alpha\langle\nabla_\theta f(\theta), p\rangle\big)$ or $\big(\|\langle p, \nabla_\theta f(\theta_i)\rangle\| > c_2\|\langle p, \nabla_\theta f(\theta)\rangle\|\big)$ then
          $\alpha \leftarrow \beta \cdot \alpha$
          if $\alpha < \epsilon$ then
              return $\alpha$
          end if
      else
          return $\alpha$
      end if
  end for
  return $\alpha$
```

C Optimizer Algorithms

Algorithm 7 Adaptive Gradient Descent

```
Input: $x_0$ (parameters to optimize), $f(x)$ (objective function)
Initialize: $\lambda_0 > 0$ (small start step), $\theta_0 \leftarrow +\infty$
  $x_1 \leftarrow x_0 - \lambda_0 \nabla f(x_0)$
  for $t = 1$ to $\ldots$ do
      $\lambda_t = \min\Big\{\sqrt{1 + \theta_{t-1}}\lambda_{t-1}, \frac{\|x_t - x_{t-1}\|}{2\|\nabla f(x_t) - \nabla f(x_{t-1})\|}\Big\}$
      $x_{t+1} \leftarrow x_t - \lambda_t \nabla f(x_t)$
      $\theta_t \leftarrow \frac{\lambda_t}{\lambda_{t-1}}$
  end for
  return $x_t$
```

Algorithm 8 Heavy Ball optimizer

```
Input: $\theta_0$ (parameters to optimize), $f(\theta)$ (objective function), $\beta$ (momentum), $\mathcal{L}(p)$ (step size choosing strategy)
  for $t = 1$ to $\dots$ do
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    if $\beta \neq 0$ then
      if $t > 1$ then
        $b_t \leftarrow \beta b_{t-1} + g_t$
      else
        $b_t \leftarrow g_t$
      end if
      $g_t \leftarrow b_t$
    end if
    $p_t \leftarrow -g_t$                                        ▷ Step direction
    Choose step size $\gamma$ according to $\mathcal{L}(p_t)$
    $\theta_t \leftarrow \theta_{t-1} + \gamma p_t$
  end for
  return $\theta_t$
```

Algorithm 9 Nesterov optimizer

```
Input: $\theta_0$ (parameters to optimize), $f(\theta)$ (objective function), $\beta$ (momentum), $\mathcal{L}(p)$ (step size choosing strategy)
  for $t = 1$ to $\dots$ do
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    if $\beta \neq 0$ then
      if $t > 1$ then
        $b_t \leftarrow \beta b_{t-1} + g_t$
      else
        $b_t \leftarrow g_t$
      end if
      $g_t \leftarrow g_t + \beta b_t$
    end if
    $p_t \leftarrow -g_t$                                        ▷ Step direction
    Choose step size $\gamma$ according to $\mathcal{L}(p_t)$
    $\theta_t \leftarrow \theta_{t-1} + \gamma p_t$
  end for
  return $\theta_t$
```

Algorithm 10 AdaGrad optimizer

```
Input: $\theta_0$ (parameters to optimize), $f(\theta)$ (objective function), $\mathcal{L}(p)$ (step size choosing strategy)
Initialize: $s_0 \leftarrow 0$ (cumulative square sum)
  for $t = 1$ to $\dots$ do
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $s_t \leftarrow \alpha s_{t-1} + g_t^2$
    $p_t \leftarrow -g_t/(\sqrt{s_t} + \epsilon)$                     ▷ Step direction
    Choose step size $\gamma$ according to $\mathcal{L}(p_t)$
    $\theta_t \leftarrow \theta_{t-1} + \gamma p_t$
  end for
  return $\theta_t$
```

Algorithm 11 RMSProp optimizer

```
**Input:** $\theta_0$ (parameters to optimize), $f(\theta)$ (objective function), $\alpha$ (alpha), $\mathcal{L}(p)$ (step size choosing strategy)
**Initialize:** $v_0 \leftarrow 0$ (square average)
  for $t = 1$ to ... do
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
      $v_t \leftarrow \alpha v_{t-1} + (1-\alpha)g_t^2$
      $p_t \leftarrow -g_t/(\sqrt{v_t} + \epsilon)$                    ▷ Step direction
      Choose step size $\gamma$ according to $\mathcal{L}(p_t)$
      $\theta_t \leftarrow \theta_{t-1} + \gamma p_t$
  end for
  return $\theta_t$
```

Algorithm 12 Adam optimizer

```
**Input:** $\theta_0$ (parameters to optimize), $f(\theta)$ (objective function), $\beta_1$, $\beta_2$ (alpha), $\mathcal{L}(p)$ (step size choosing strategy)
**Initialize:** $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment)
  for $t = 1$ to ... do
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
      $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
      $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
      $\hat{m}_t \leftarrow m_t/(1-\beta_1^t)$
      $\hat{v}_t \leftarrow v_t/(1-\beta_2^t)$
      $p_t \leftarrow -\hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$                    ▷ Step direction
      Choose step size $\gamma$ according to $\mathcal{L}(p_t)$
      $\theta_t \leftarrow \theta_{t-1} + \gamma p_t$
  end for
  return $\theta_t$
```

Algorithm 13 BFGS optimizer

```
**Input:** $\theta_0$ (parameters to optimize), $f(\theta)$ (objective function), $\mathcal{L}(p)$ (step size choosing strategy)
**Initialize:** $H_0 \leftarrow I$
  for $t = 1$ to ... do
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
      $p_t \leftarrow -H_{k-1}g_t$                    ▷ Step direction
      Choose step size $\gamma$ according to $\mathcal{L}(p_t)$          ▷ $\gamma$ should satisfy Wolfe conditions
      $s_t \leftarrow \gamma p_t$
      $\theta_t \leftarrow \theta_{t-1} + s_t$
      $y_t \leftarrow \nabla_\theta f_t(\theta_t) - g_t$
      $H_t \leftarrow H_{t-1} + \frac{(s_t^T y_t + y_t^T H_{t-1} y_t)(s_t s_t^T)}{(s_t^T y_t)^2} - \frac{H_{t-1} y_t s_t^T + s_t y_t^T H_{t-1}}{s_t^T y_t}$
  end for
  return $\theta_t$
```

D Experiments

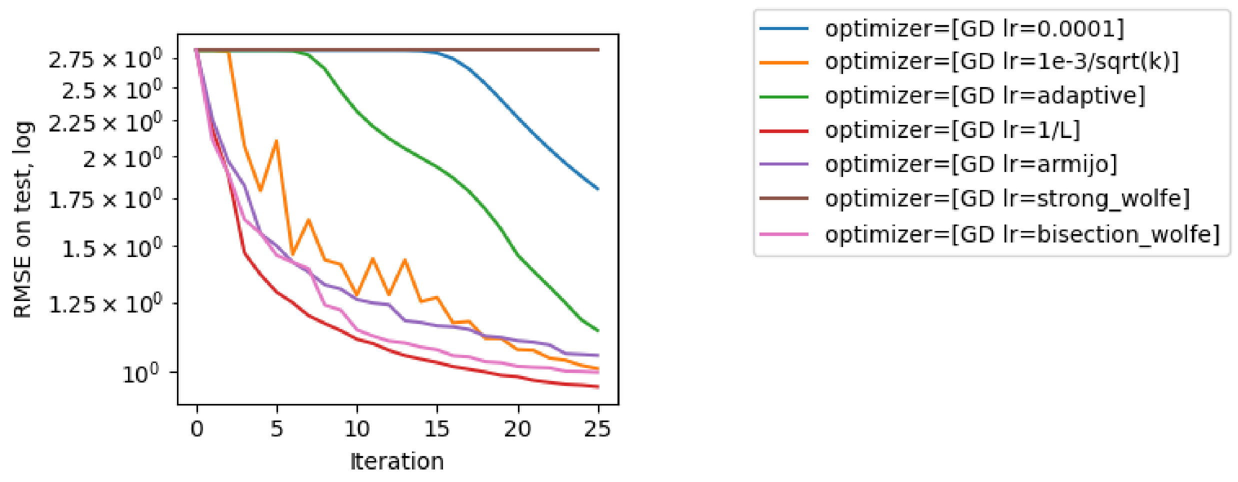

Figure 5: Gradient descent (Algorithm 1)

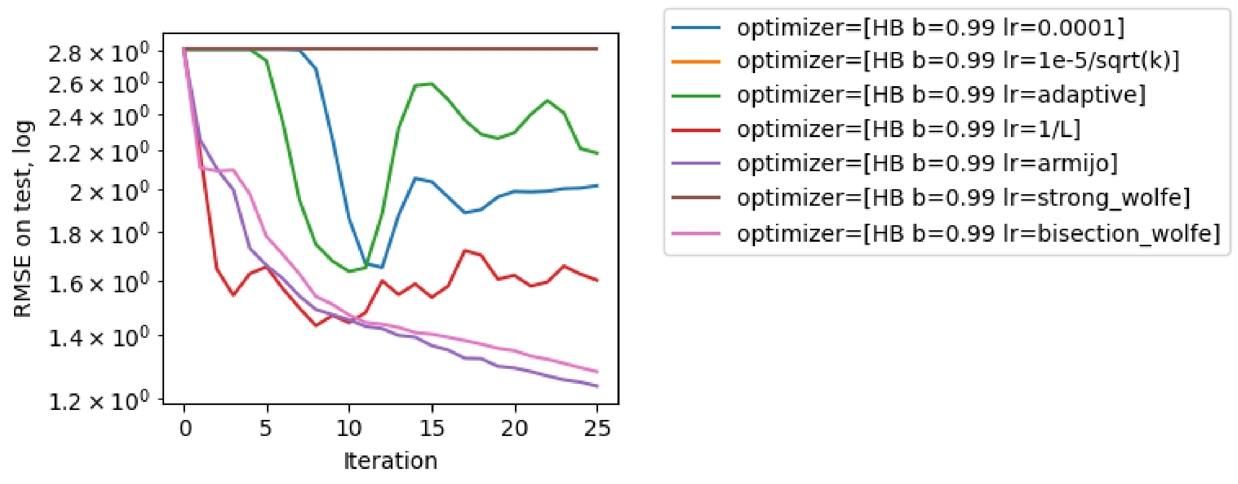

Figure 6: Heavy Ball (Algorithm 8)

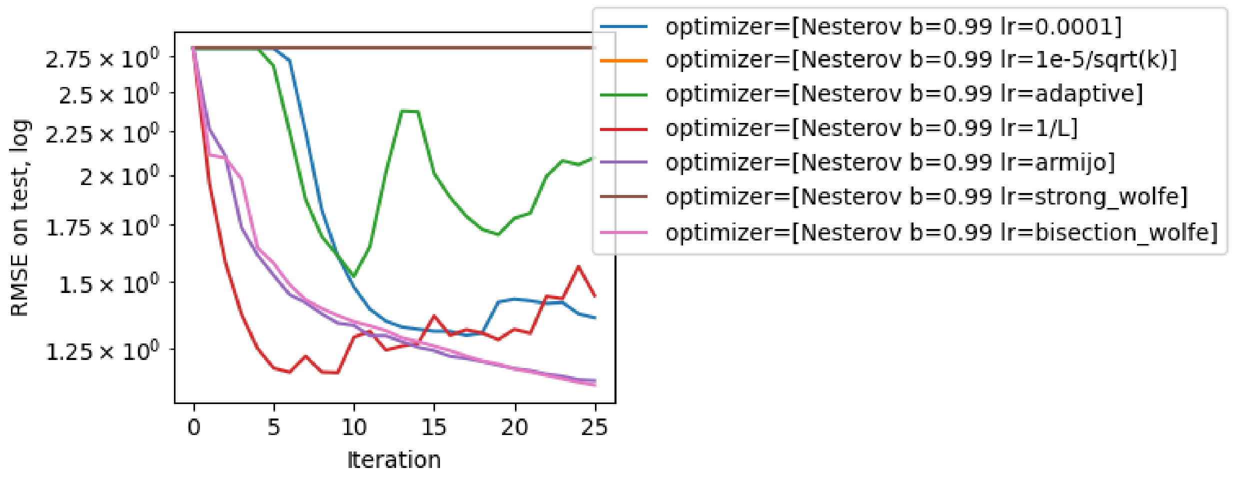

Figure 7: Nesterov Accelerated Gradient (Algorithm 9)

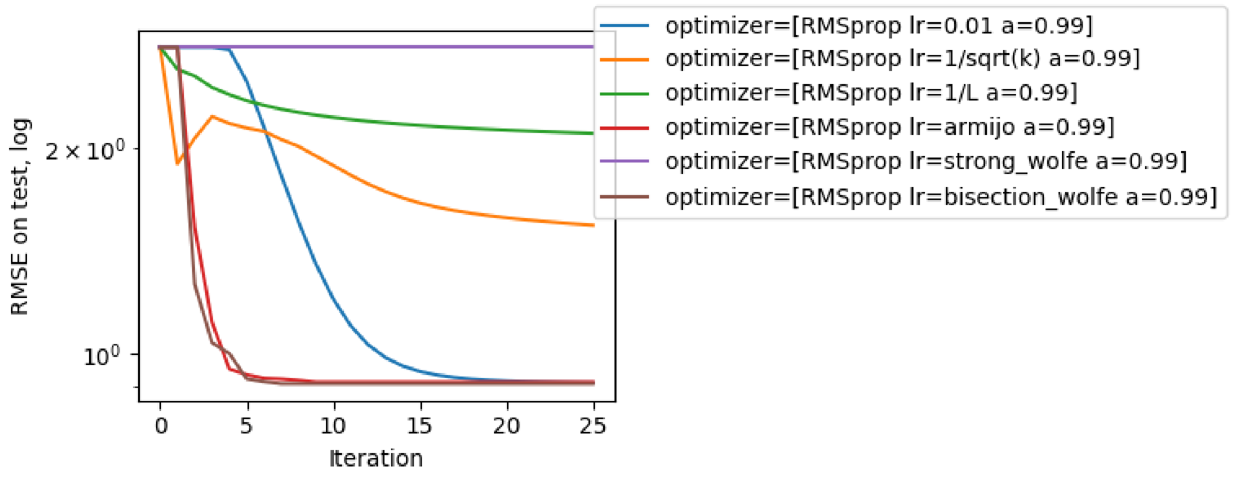

Figure 8: RMSProp (Algorithm 11)

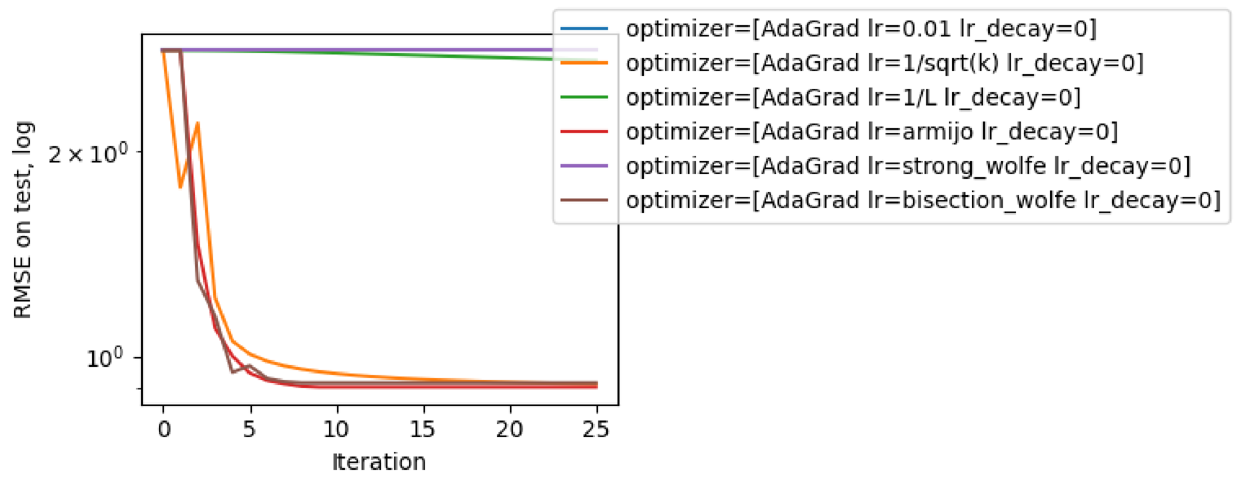

Figure 9: AdaGrad (Algorithm 10)

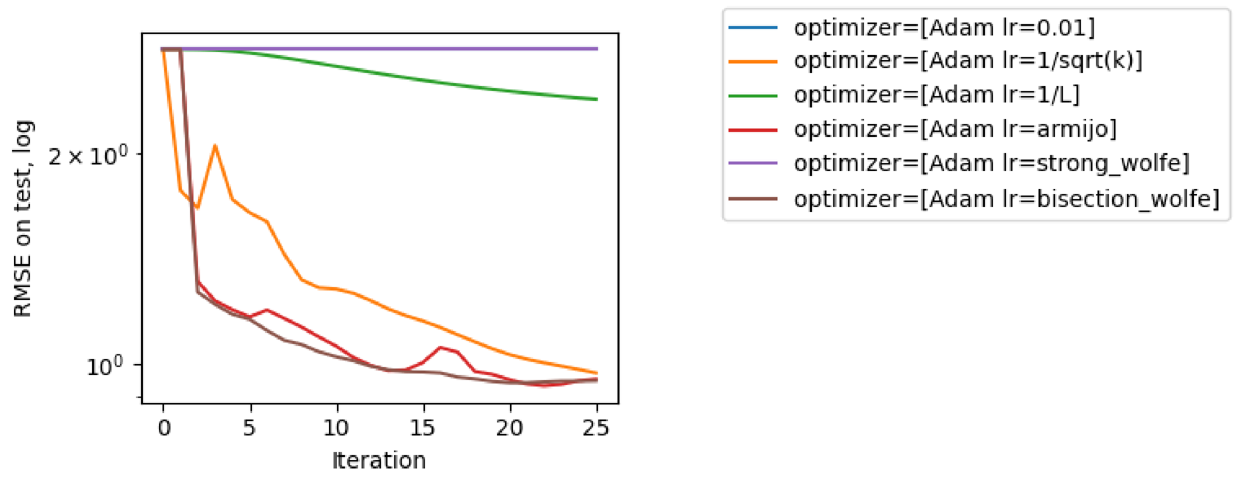

Figure 10: Adam (Algorithm 12)

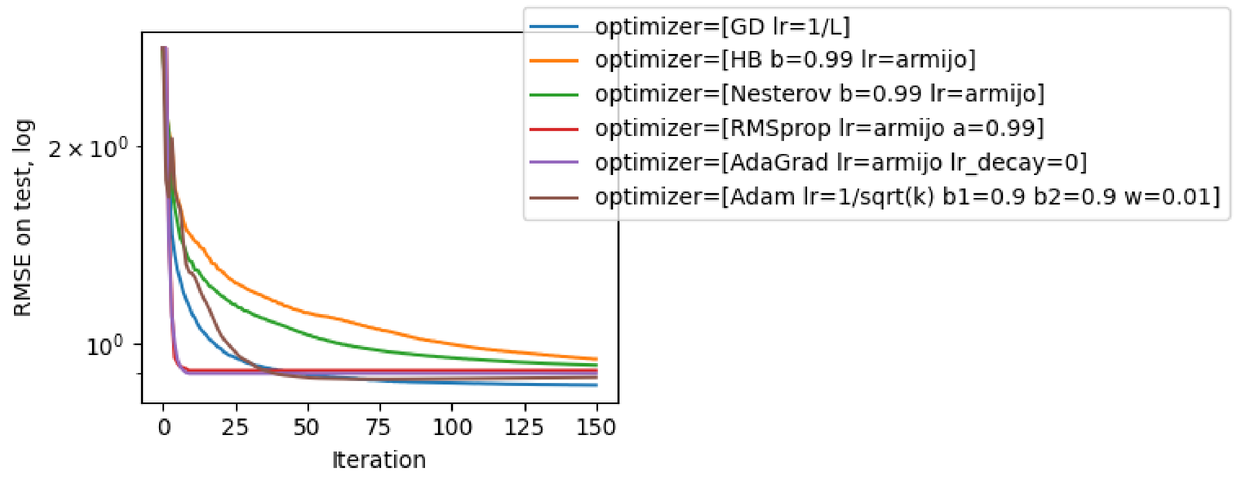

Figure 11: Comparison of optimizers with best configurations

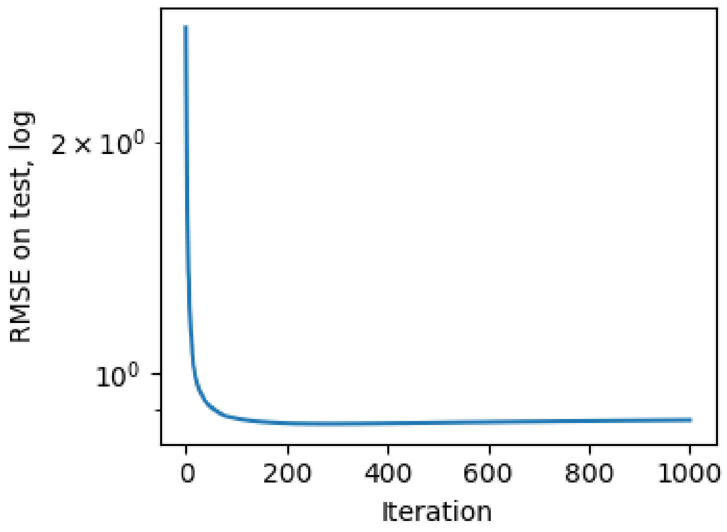

Figure 12: Best model: $r = 10$, Gradient Descent with estimate 1/L strategy and $\lambda = 2$

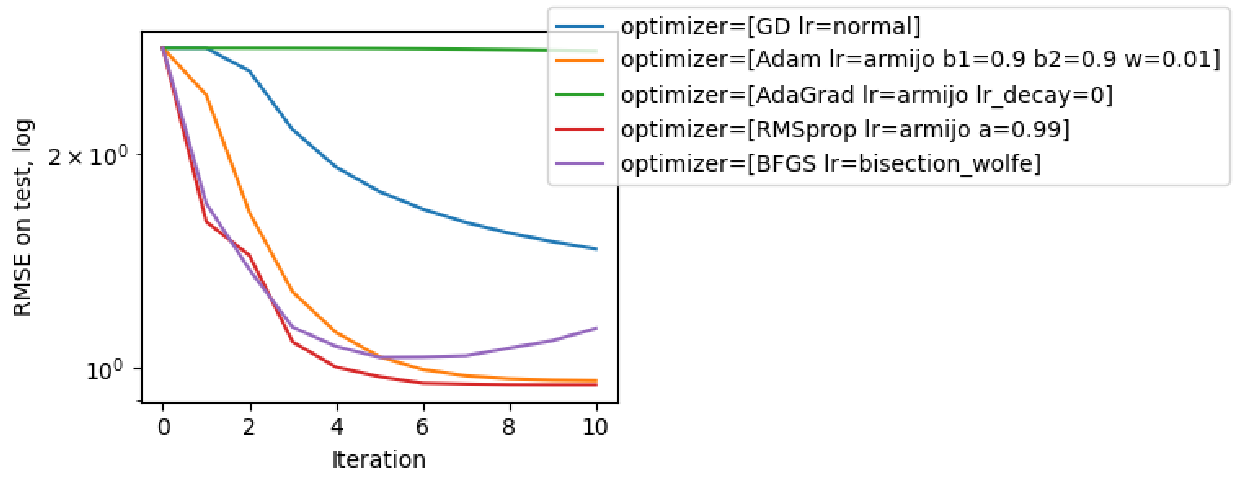

Figure 13: Vector Gradient Descent

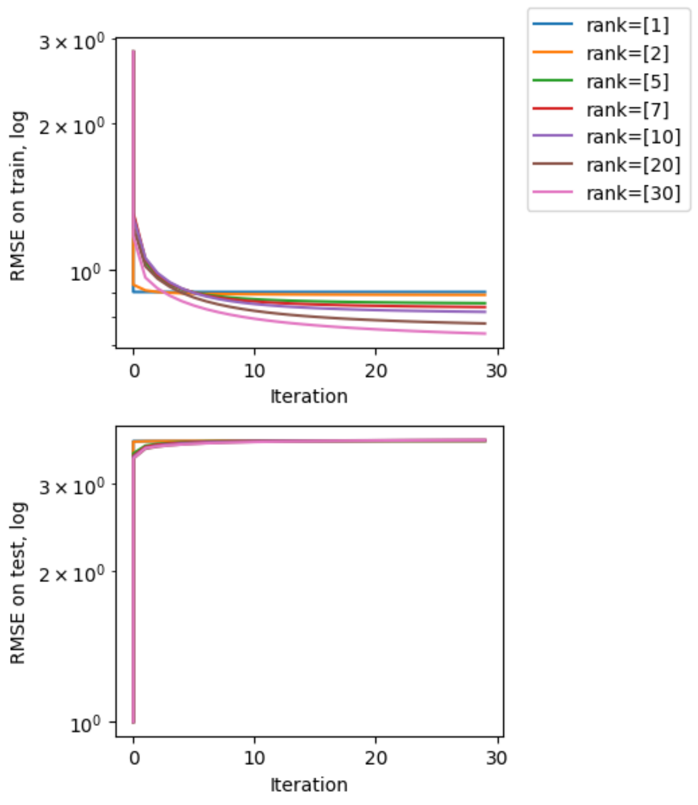

Figure 14: Non-Negative Matrix Factorization