Quadratic Programming with a Neural Network

Keith Dillon

November 27, 2019

Abstract

This note describes how to implement and solve a quadratic programming optimization problem using a shallow neural network in Keras. A single linear layer is used with a custom one-sided loss to impose the inequality constraints. A custom kernel regularizer is used to impose the optimization objective, yielding a form of penalty method. This provides a useful exercise in augmenting the loss, metrics, and callbacks used in Keras. This also potentially allows the exploitation of the back-end implementations of Keras and Tensorflow on GPU's and distributed storage.

Introduction

A general quadratic program (QP) is commonly written in the form [1]:

$$\begin{aligned} \min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{P}\mathbf{w} + \mathbf{q}^T\mathbf{w} \\ \mathbf{G}\mathbf{w} \leq \mathbf{h} \\ \mathbf{A}\mathbf{w} = \mathbf{b} \end{aligned} \tag{1}$$

where $\mathbf{P}$, $\mathbf{G}$, and $\mathbf{A}$ are matrices, and $\mathbf{q}$, $\mathbf{h}$, $\mathbf{b}$ and $\mathbf{w}$ are vectors; we used the letter $\mathbf{w}$ for the free variable to be optimized over, for consistency with the artificial neuron notation we will use. By replacing the equality constraint $\mathbf{A}\mathbf{w} = \mathbf{b}$ with inequality constraints $\mathbf{A}\mathbf{w} \leq \mathbf{b}$ and $\mathbf{A}\mathbf{w} \geq \mathbf{b}$ (the latter can be written equivalently as $-\mathbf{A}\mathbf{w} \leq -\mathbf{b}$), we can augment the $\mathbf{G}$ and $\mathbf{h}$ definitions appropriately to use the simpler description,

$$\begin{aligned} \min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{P}\mathbf{w} + \mathbf{q}^T\mathbf{w} \\ \mathbf{G}\mathbf{w} \leq \mathbf{h} \end{aligned} \tag{2}$$

A penalty method solves this problem by replacing it with problems of the form

$$\min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{P}\mathbf{w} + \mathbf{q}^T\mathbf{w} + \phi(\mathbf{G}\mathbf{w} - \mathbf{h}) \tag{3}$$

The so-called auxiliary function ϕ is designed to produce a large value when the constraint is violated [1], as in

$$\phi(\mathbf{G}\mathbf{w} - \mathbf{h}) = \begin{cases} large, & \mathbf{g}_i^T\mathbf{w} - h_i > 0 \text{ for any } i \\ 0, & \mathbf{g}_i^T\mathbf{w} - h_i \leq 0 \text{ for all } i \end{cases} \tag{4}$$

where $\mathbf{g}_i^T$ is the ith row of $\mathbf{G}$ and h_i is the ith element of $\mathbf{h}$. Minimization of this term forces the solution toward fulfilling the constraints.

Meanwhile, an artificial neuron implements the function

$$y = f(\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{x} + b), \tag{5}$$

where σ is the activation function; $\mathbf{w}$ is the (to be learned) weight vector; b is the (to be learned) bias. $\mathbf{x}$

refers to an input sample and y is the prediction or target for that sample. The learning (also called training) in frameworks such as Keras[1][2] is performed by some variant of stochastic gradient descent applied to a loss function $L(f(\mathbf{x};\mathbf{w}), y_{true})$ for a set of data $\{(\mathbf{x}^{(1)}, y^{(1)}_{true}), (\mathbf{x}^{(2)}, y^{(2)}_{true}), ..., (\mathbf{x}^{(K)}, y^{(K)}_{true})\}$. Our goal, therefore is to formulate the optimization problem of Eq. (3) using this framework. In other words, we wish to learn weights which solve the quadratic program.

It is possible to impose constraints on the weights in Keras[2], including non-negativity and norms constraints. Hence some simple optimization problem can be directly implemented with a linear layer (i.e. no activation function). In this note we show how to perform a full implementation of the general form of the problem of Eq. (3), both for its instructive value in extending Keras methods, as well as for potential use in implementing large optimization problems on such frameworks.

Methods

In order to adapt Keras to a general quadratic programming problem, we will employ the follow process:

1. Pass rows $\mathbf{g}_i^T$ of $\mathbf{G}$ in as input samples $\mathbf{x}^{(i)}$, and corresponding elements h_i of $\mathbf{h}$ as targets $y^{(i)}$.
2. Create a network consisting of a single node with no bias or activation, to compute the product $\mathbf{g}_i^T\mathbf{w}$.
3. Use a custom one-sided loss function to penalize violations of the QP inequalities $\mathbf{g}_i^T\mathbf{w} \leq h_i$.
4. Use a custom weight regularization function $R(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{P}\mathbf{w} + \mathbf{q}^T\mathbf{w}$ to impose the QP objective.
5. Successively decrease a scalar weight on $R(\mathbf{w})$ to achieve a (relatively) increasing penalty on constraint violations.

In this approach, the constraint matrix $\mathbf{G}$ may be arbitrarily large, as it is only used a row (or small batch of rows) at a time; it never need be stored in its entirety in local memory . The objective matrix $\mathbf{P}$, on the other hand, must be stored in memory for use.

Constraints via Loss

First we provide a function to create a custom one-sided loss. In our case, we want to implement a penalty function which only penalizes outputs which violate the constraint. There are many possible options for this, used as penalty and barrier methods in optimization research. Here we will use a one-sided penalty as follows:

$$L(f(\mathbf{x}), y) = \begin{cases} \big(f(\mathbf{x};\mathbf{w}) - y\big)^2, & f(\mathbf{x};\mathbf{w}) > y \\ 0, & f(\mathbf{x};\mathbf{w}) \leq y \end{cases} \tag{6}$$

Creating custom losses can be tricky in Keras; the function must operate over Keras tensors, utilizing a limited set of available operators. For example the multiplication operator (as of this writing) cannot directly multiply a float32 number by a float16 number, requiring explicit casting. If using Keras with Tensorflow (e.g., using the Keras included with tensorflow), custom functions are created using functions linked in the backend library[3].

A python implementation of this loss function is:

```
def one_sided_l2(y_tar, y_pred):
    diff = y_pred - y_tar
    mdiff = diff>0 # mask of positive diff, satisfies constraint
    mdiff32 = keras.backend.cast(mdiff,'float32') # need float32 for product below
    return keras.backend.mean(keras.backend.square(mdiff32*diff), axis=-1)
```

[1] https://keras.io/
[2] https://keras.io/constraints/
[3] https://www.tensorflow.org/api_docs/python/tf/keras/backend

Next we will demonstrate the use of our one-sided loss to find a feasible solution, i.e. in the set of possible solutions which fulfill the constraints,

$$S = \{\mathbf{w} : \mathbf{G}\mathbf{w} \leq \mathbf{h}\}. \tag{7}$$

In the following code we create a simple model using a single-node network to find a feasible solution, i.e., a member of the set S,

```
import tensorflow as tf
from tensorflow import keras

G = numpy.asarray([[-1.0,0.0,-1.0,2.0,3.0],[0.0,-1.0,-3.0,5.0,4.0]]).T
h = numpy.asarray([0.0,0.0,-15.0,100.0,80.0]).T

model = keras.Sequential()
model.add(keras.layers.Dense(1, input_dim=2, use_bias=False))
sgd = keras.optimizers.SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd,loss=one_sided_l2,metrics=['mse'])
history = model.fit(G, h, epochs=5)
```

This code implements the function $f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^T\mathbf{x}$ with the single layer. It operates by computing the loss on a batch-wise basis, computing the loss for batch of rows of the constraints at a time, as in $\sum_{i \in \text{batch}} L(\mathbf{w}^T\mathbf{g}_i, h_i)$. This loss is minimized using gradient with respect to the weights $\mathbf{w}$ for each batch.

As shown in Fig. 1, the model converges to a feasible point quickly. To get the solution, extract the weights from the model as in the code `w = model.get_weights()`.

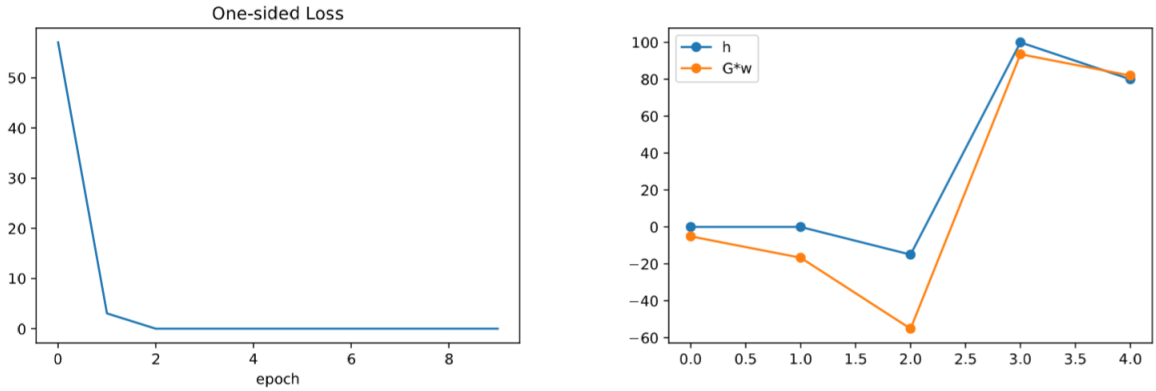

Figure 1: Optimization problem #1 (feasibility problem): (left) Loss function versus epoch; (right) $\mathbf{G}\mathbf{w}$ versus $\mathbf{h}$, demonstrating fulfilled constraints $\mathbf{G}\mathbf{w} \leq \mathbf{h}$.

Objectives via Regularization

Next we will formulate the QP objective with weight regularization Weight regularization is implemented by adding a term to the loss, as in

$$L_R(f(\mathbf{x}; \mathbf{w}), y) = L(f(\mathbf{x}; \mathbf{w}), y) + \lambda R(\mathbf{w}), \tag{8}$$

The parameter λ is a non-negative hyperparameter we must choose to trade off the relative importance of the two terms. We will return to this later.

In the present version of Keras, the term "regularizers" refer to regularization imposed on the values in individual layers[4]. The weight regularization option in particular is called "kernel_regularizer". Other regularizer options are "bias_regularizer" (imposed on bias values), and "activity_regularizer" (imposed on layer outputs). Built-in

[4] https://keras.io/regularizers/

options for kernel regularization are ℓ_1 (to impose sparsity of $\mathbf{w}$), ℓ_2 (also known as weight decay), and ℓ_1-ℓ_2 (elastic net). For example, $R(\mathbf{w}) = \|\mathbf{w}\|_2^2$ in Eq. (8) for the ℓ_2 option. These simple regularizers, when used along with our one-sided norm, can directly provide a norm objectives, such as the following optimization problem,

$$\min_{\mathbf{w}} \|\mathbf{w}\| \quad \mathbf{G}\mathbf{w} \leq \mathbf{h} \tag{9}$$

By minimizing the loss of Eq. (8), we enforce the inequality constraint by minimizing the first term, and minimize the objective by minimizing the second term. A variety of interesting problems can be formulated with these built-in regularizer options.

As with loss functions, regularizers can also be extended with custom functions. To impose the general quadratic programming objective, $\frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x}$, we define the following function to use as kernel regularizer,

```
def xPx_qx(x):
    xPx = keras.backend.transpose(x)@P@x
    qx = keras.backend.transpose(keras.backend.cast(q,'float32'))@x
    return lambda_reg*(0.5*xPx+qx)
```

Note that the "@" operator performs matrix multiplication, which in this tensor implementation appears to be completely equivalent to `matmul` and `dot`. For example, even when using `keras.backend.dot` for a pair of column vectors, we must transpose the first vector ourselves. We can impose this regularizer on the layer as follows:

```
model.add(keras.layers.Dense(1, input_dim=1000, use_bias=False, kernel_regularizer=
    xPx_qx))
```

Monitoring Optimization with Callbacks

In optimization, as in training machine learning models, it is important to monitor the performance during the initial stages of application to a problem, to aid in choosing parameters and settings. In the implementation given thus far, we will only see the intermediate loss function calculations, not the objective. In order to also monitor the objective value during optimization, we must create a custom callback. A callback[5] is a function which the framework can be programmed to call at desired times, such as after every epoch in training The following code saves the intermediate objective calculations for plotting,

```
def QP_metric(model):
    w = model.layers[0].get_weights()[0]
    return xPx_qx(w)

class ObjHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.objective = []

    def on_epoch_end(self, batch, logs={}):
        self.objective.append(QP_metric(model).numpy()[0][0])

objhist = ObjHistory()
```

To print the objective value at each epoch, we include the following callback as well:

```
from tensorflow.keras.callbacks import LambdaCallback
print_obj = LambdaCallback(on_epoch_end=lambda batch, logs: print(QP_metric(model)))
```

The following example demonstrates the use of our custom regularizer and callbacks

```
P = numpy.asarray([[1.,2.],[0.,4.]])
q = numpy.asarray([[1.],[-1.]])
G = numpy.asarray([[-1.0,  0.0, -1.0, 2.0, 3.0],
```

[5] https://keras.io/callbacks/

```
[ 0.0, -1.0, -3.0, 5.0, 4.0]]).T
h = numpy.asarray([0.0, 0.0, -15.0, 100.0, 80.0]).T

model = keras.Sequential()
model.add(keras.layers.Dense(1, input_dim=2, use_bias=False, kernel_regularizer=
    xPx_qx))

sgd = keras.optimizers.SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)

lambda_reg=1e-6

model.compile(optimizer=sgd,
loss=one_sided_l2,
metrics=['mse'])

model.fit(G.reshape(5,2), h.reshape(5,), epochs=1000, callbacks = [print_QP_metric,
    objhist])
```

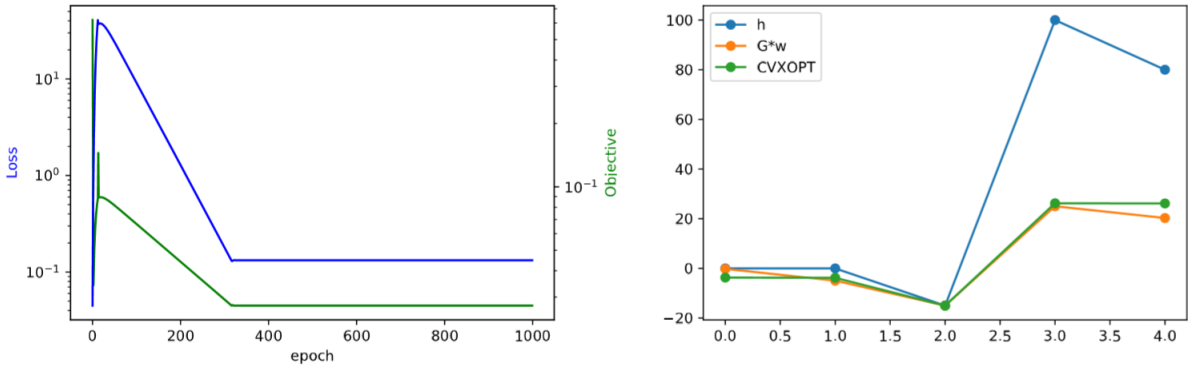

Figure 2: Optimization problem #1 (feasibility problem): (left) Loss function versus epoch; (right) $\mathbf{Gw}$ versus $\mathbf{h}$, demonstrating fulfilled constraints $\mathbf{Gw} \leq \mathbf{h}$.

The results are given in Fig. 2, and compared to the result computed by CVXOPT [3] for this problem (see Appendix A). In this case an optimal objective value of 44.85 was achieved by the Keras implementation, versus an optimal of 49.07 achieved by CVXOPT. The different objective value is achieved by taking advantage of slight constraint violations (too small to see in Fig. 2, which allow a lower objective value to be found. To more strictly enforce the constraints we must increase the relative weighting of the penalty term, which we address next.

Iteratively Increasing the Penalty

Finally we provide an example which iteratively increases the relative weight of the penalty term (enforcing the inequality constraint), by reducing the scaling on the regularizer. In the following code `lambda_reg` is a global variable used in the `xPx_qx` function to scale the regularization term. By reducing this variable at each pass through the loop, we increase the relative importance of the penalty term.

```
model = keras.Sequential()
model.add(keras.layers.Dense(1, input_dim=1000, use_bias=False, kernel_regularizer=
    xPx_qx))
model.summary()

lambda_reg = 1e-3

sgd = keras.optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(optimizer=sgd,
```

```
loss=one_sided_l2,
metrics=['mse'])

Histo = {}
for k_outer in range(5):
    model = keras.Sequential()
    model.add(keras.layers.Dense(1, input_dim=1000, use_bias=False,
        kernel_regularizer=xPx_qx))

    lambda_reg = lambda_reg/10.

    if k_outer>0:
        model.set_weights(w)

    model.compile(optimizer=sgd, loss=one_sided_l2, metrics=['mse'])
    Histo[k_outer] = model.fit(G, h, epochs=100, callbacks = [print_QP_metric,objhist
        ], batch_size=128, shuffle=True)
    w = model.get_weights()
```

The results are shown in Fig. 3. The final objective value achieved was 479730, compared to 477156 achieved

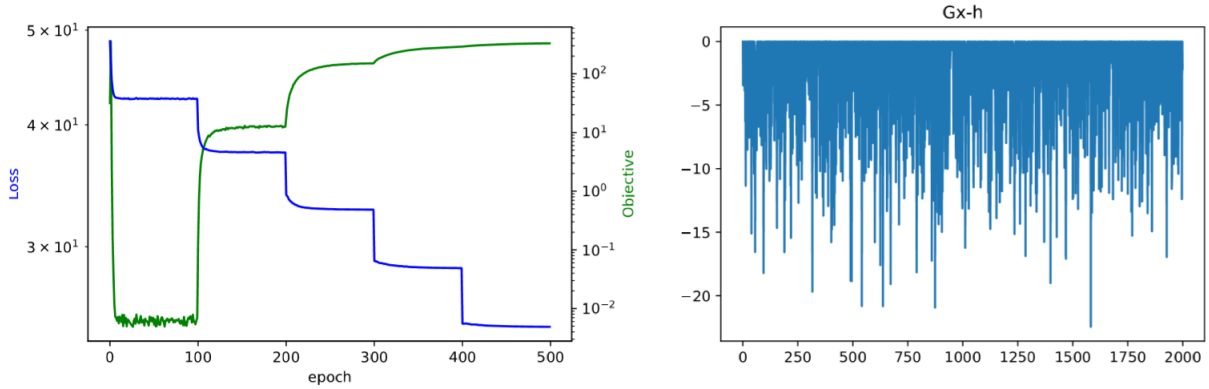

Figure 3: Optimization problem #1 (feasibility problem): (left) Loss function versus epoch; (right) $\mathbf{Gw}$ versus $\mathbf{h}$, demonstrating fulfilled constraints $\mathbf{Gw} \leq \mathbf{h}$.

by CVXOPT (Appendix B).

Discussion

We have demonstrated how a quadratic programming solver may be implemented by customizing a neural network framework such as Keras. This suggests some interesting ideas for exploiting the hardware implementations on which such software packages are implemented. In the simple examples shown in this note, a GPU implementation would be overkill. Even in a much larger-scale case, optimization time would likely be dominated by data transfer, as the actual network complexity is minimal. However for many problems, the constraint matrix may be implement in an online fashion directly on the GPU. An example might be a very large image reconstruction application, where the forward model of the physical propagation of radiation is simulated directly on the GPU. In this case only basic parameters, such as view direction and collected sensor data, would need to be transferred to the GPU.

An open problem with both optimization solvers as well as machine learning platforms is the choice of optimization parameters (i.e. hyperparameters and settings) which provide the fastest convergence. Parameters such as learning rate and momentum can cause total optimization time to vary by orders of magnitude, making the difference between a solvable and unsolvable problem when dealing with extremely large-scale problems. A new perspective such as provided here, which relates these different fields, would hopefully lead to new insights.

Appendix A: CVXOPT Example 1

```
import numpy
from cvxopt import matrix
from cvxopt import solvers

P = numpy.asarray([[1.,2.],[0.,4.]])
q = numpy.asarray([[1.],[-1.]])
G = numpy.asarray([[-1.0,  0.0, -1.0, 2.0, 3.0],
[ 0.0, -1.0, -3.0, 5.0, 4.0]]).T
h = numpy.asarray([0.0, 0.0, -15.0, 100.0, 80.0]).T

sol = solvers.qp(matrix(P, tc='d'),
                 matrix(q, tc='d'),
                 matrix(G, tc='d'),
                 matrix(h, tc='d'))

     pcost       dcost       gap    pres   dres
0:  2.8110e+02 -2.6664e+02  5e+02  6e-17  1e+02
1:  2.4798e+02  1.4789e+02  1e+02  8e-17  2e+01
2:  4.8707e+01 -1.7918e+02  2e+02  1e-16  2e+01
3:  3.5331e+01 -5.7893e+01  9e+01  4e-16  4e+00
4:  3.5231e+01  3.3978e+01  1e+00  1e-17  4e-02
5:  3.5155e+01  3.5142e+01  1e-02  1e-16  4e-04
6:  3.5154e+01  3.5154e+01  1e-04  2e-16  4e-06
7:  3.5154e+01  3.5154e+01  1e-06  8e-17  4e-08
Optimal solution found.

sol
{'x': <2x1 matrix, tc='d'>,
'y': <0x1 matrix, tc='d'>,
's': <5x1 matrix, tc='d'>,
'z': <5x1 matrix, tc='d'>,
'status': 'optimal',
'gap': 1.2617910893651018e-06,
'relative gap': 3.58934020023423e-08,
'primal objective': 35.15384622980947,
'dual objective': 35.15384496801838,
'primal infeasibility': 7.538017664871812e-17,
'dual infeasibility': 4.2467004103447406e-08,
'primal slack': 1.618890391498939e-08,
'dual slack': 1.6445441148643891e-09,
'iterations': 7}
```

Appendix B: CVXOPT Example 2

```
numpy.random.seed(0)
P = numpy.random.randn(1000,1000)
P = P@P.T # make positive semidefinite (Convex QP)
q = numpy.random.randn(1000,1)
x_true = numpy.random.randn(1000,1)
slack = numpy.random.randn(2000,1)
G = numpy.random.randn(2000,1000)
h = G@x_true + slack*(slack>0)

sol = solvers.qp(matrix(P, tc='d'),
                 matrix(q, tc='d'),
                 matrix(G, tc='d'),
                 matrix(h, tc='d'))
```

```
     pcost       dcost       gap    pres   dres
 0:  1.2774e+05  3.1594e+05  2e+06  1e+00  1e+03
 1:  2.2203e+05  1.4833e+05  1e+06  4e-01  7e+02
 2:  3.2562e+05  2.9089e+05  5e+05  2e-01  3e+02
 3:  3.9010e+05  3.7999e+05  3e+05  8e-02  1e+02
 4:  4.3268e+05  4.3003e+05  1e+05  4e-02  5e+01
 5:  4.5734e+05  4.4796e+05  9e+04  2e-02  3e+01
 6:  4.7088e+05  4.6377e+05  4e+04  7e-03  1e+01
 7:  4.7615e+05  4.7193e+05  1e+04  2e-03  3e+00
 8:  4.7736e+05  4.7509e+05  5e+03  6e-04  9e-01
 9:  4.7754e+05  4.7652e+05  1e+03  7e-05  1e-01
10:  4.7730e+05  4.7694e+05  4e+02  2e-05  2e-02
11:  4.7718e+05  4.7713e+05  6e+01  2e-06  2e-03
12:  4.7716e+05  4.7715e+05  5e+00  1e-07  2e-04
13:  4.7716e+05  4.7716e+05  8e-02  2e-09  3e-06
14:  4.7716e+05  4.7716e+05  1e-03  2e-11  4e-08
Optimal solution found.

sol
{'x': <1000x1 matrix, tc='d'>,
'y': <0x1 matrix, tc='d'>,
's': <2000x1 matrix, tc='d'>,
'z': <2000x1 matrix, tc='d'>,
'status': 'optimal',
'gap': 0.0010098798005411858,
'relative gap': 2.1164543557666675e-09,
'primal objective': 477156.42897668254,
'dual objective': 477156.42805600003,
'primal infeasibility': 2.196039044268912e-11,
'dual infeasibility': 3.5348859757334116e-08,
'primal slack': 9.306128382129809e-10,
'dual slack': 2.7110081519145572e-08,
'iterations': 14}
```

References

[1] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004.

[2] Francois Chollet. *Deep Learning with Python*. Manning Publications Company, October 2017. Google-Books-ID: Yo3CAQAACAAJ.

[3] Joachin Dahl and Lieven Vandenberghe. Cvxopt: A python package for convex optimization. In *Proc. eur. conf. op. res*, volume 2, page 3, 2006.