stochqn Documentation

David Cortes

Apr 15, 2021

Contents:

class SQN(*x0*, *grad_fun*, *obj_fun=None*, *hess_vec_fun=None*, *pred_fun=None*, *batches_per_epoch=25*, *step_size=0.001*, *decr_step_size='auto'*, *shuffle_data=True*, *random_state=1*, *nepochs=25*, *valset_frac=None*, *tol=0.1*, *callback_epoch=None*, *callback_iter=None*, *kwargs_cb={}*, *verbose=True*, *mem_size=10*, *bfgs_upd_freq=20*, *min_curvature=0.0001*, *y_reg=None*, *use_grad_diff=False*, *check_nan=True*, *nthreads=-1*, *use_float=False*)

SQN optimizer

Optimizes an empirical (convex) loss function over batches of sample data.

Parameters

- **x0** (*array (m,)*) – Initial values of the variables to optimize (refered hereafter as 'x').
- **grad_fun** (*function(x, X, y, sample_weight, **kwargs) –> array(m,)*) – Function that calculates the empirical gradient at values 'x' on data 'X' and 'y'. Note: output must be one-dimensional and with the same number of entries as 'x', otherwise the Python session might segfault. (The extra keyword arguments are passed in the 'fit' method, not here)
- **obj_fun** (*function(x, X, y, sample_weight, **kwargs) –> float*) – Function that calculates the empirical objective value at values 'x' on data 'X' and 'y'. Only used when using a validation set ('valset_frac' not None, or 'valset' passed to fit). Ignored when fitting the data in user-provided batches. (The extra keyword arguments are passed in the 'fit' method, not here)
- **hess_vec_fun** (*function(x, vec, X, y, sample_weight, **kwargs) –> array(m,)*) – Function that calculates the product of a vector the empirical Hessian at values 'x' on data 'X' and 'y'. Ignored when using 'use_grad_diff=True'. Note: output must be one-dimensional and with the same number of entries as 'x', otherwise the Python session might segfault. These products are calculated on a larger batch than the gradients (given by batch_size * bfgs_upd_freq). (The extra keyword arguments are passed in the 'fit' method, not here)
- **pred_fun** (*None or function(xopt, X)*) – Prediction function taking as input the optimal 'x' values as obtained by the optimization procedure, and new observation 'X' on which to make predictions. If passed, will have an additional method oLBFGS.predict(X, *args) that calls this function with current values of 'x'.
- **batches_per_epoch** (*int*) – Number of batches per epoch (each batch will have the same number of observations except for the last one which might be smaller).
- **step_size** (*float*) – Initial step size to use. (Can be modified after object is already initialized)
- **decr_step_size** (*str "auto", None, or function(initial_step_size, epoch) -> float*) – Function that determines the step size during each epoch, taking as input the initial step size and the epoch number (starting at zero). If "auto", will use 1/sqrt(iteration). If None, will use constant step size. For 'partial_fit', it will take as input the number of iterations of the algorithm rather than epoch, so it's very recommended to provide a custom function when passing data in user-provided batches. Can be modified after the object has been initialized (oLBFGS.decr_step_size)
- **shuffle_data** (*bool*) – Whether to shuffle the data at the beginning of each epoch.
- **random_state** (*int*) – Random seed to use for shuffling data and selecting validation set. The algorithm is deterministic so it's not used for anything else.
- **nepochs** (*int*) – Number of epochs for which to run the optimization procedure. Might terminate earlier if using a validation set for monitoring.
- **valset_frac** (*float(0, 1) or None*) – Percent of the data to use as validation set for early stopping. Can also pass a user-provided validation set to 'fit', in which case it will be ignored. If passing None, will run for the number of epochs passed in 'nepochs'.

- **tol** (*float*) – If the objective function calculated on the validation set decrease by less than 'tol' upon completion of an epoch, will terminate the optimization procedure. Ignored when not using a validation set.
- **callback_epoch** (*None or function*(x, **kwargs)*) – Callback function to call at the end of each epoch
- **callback_iter** (*None or function*(x, **kwargs)*) – Callback function to call at the end of each iteration
- **kwargs_cb** (*tuple*) – Additional arguments to pass to 'callback' and 'stop_crit'. (Can be modified after object is already initialized)
- **verbose** (*bool*) – Whether to print messages when there is some problem during an iteration (e.g. correction pair not meeting minum curvature).
- **mem_size** (*int*) – Number of correction pairs to store for approximation of Hessian-vector products.
- **bfgs_upd_freq** (*int*) – Number of iterations (batches) after which to generate a BFGS correction pair.
- **min_curvature** (*float or None*) – Minimum value of s*y / s*s in order to accept a correction pair.
- **y_reg** (*float or None*) – regularizer for 'y' vector (gets added y_reg * s)
- **use_grad_diff** (*bool*) – Whether to create the correction pairs using differences between gradients instead of Hessian-vector products. These gradients are calculated on a larger batch than the regular ones (given by batch_size * bfgs_upd_freq).
- **check_nan** (*bool*) – Whether to check for variables becoming NaN after each iteration, and reverting the step if they do (will also reset BFGS memory).
- **nthreads** (*int*) – Number of parallel threads to use. If set to -1, will determine the number of available threads and use all of them. Note however that not all the computations can be parallelized.
- **use_float** (*bool*) – Whether to use C 'float' type (np.float32). If 'False' (the default), will use 'double' type (np.float64). The variables, gradient, and hessian-vector must be of this same dtype.

References

fit (*X*, *y*, *sample_weight=None*, *additional_kwargs={}*, *valset=None*)
Fit model to sample data

Parameters

- **X** (*array(n_samples, m)*) – Sample data to which to fit the model.
- **y** (*array(n_samples,)*) – Labels or target values for the sample data.
- **sample_weight** (*None or array(n_samples,)*) – Observations weights for the sample data.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to the objective, gradient, and Hessian-vector functions.
- **valset** (*tuple(3)*) – User-provided validation set containing (X_val, y_val, sample_weight_val). At the end of each epoch, will calculate objective function on this set, and if the decrease from the objective function in the previous epoch is below tolerance, will terminate procedure earlier. If 'valset_frac' was provided and a validation set is passed,

'valset_frac' will be ignored. Must provide objective function in order to use a validation set.

Returns **self** – This object.

Return type obj

get_x()
Get a copy of current values of the variables

Returns **x** – Current variable values.

Return type array(n,)

niter

partial_fit(*X*, *y*, *sample_weight=None*, *additional_kwargs={}*)
Update model with user-provided batches of data

Note: In SQN and adaQN, the data passed to all calls in partial fit will be stored in a limited-memory container which will be used to calculate Hessian-vector products or large-batch gradients. The size of this container is determined by the inputs 'batch_size' and 'bfgs_upd_freq' passed in the constructor call.

Note: The step size in partial fit is determined by the number of optimizer iterations rather than the number of epochs, thus for a given amount of data, the default step size will be much smaller than when calling 'fit'. Recommended to provide a custom step size function ('decr_step_size' in the initialization), as otherwise the step size sequence will be too small.

Parameters

- **X** (*array(n_samples, m)*) – Sample data to with which to update the model.
- **y** (*array(n_samples,)*) – Labels or target values for the sample data.
- **sample_weight** (*None or array(n_samples,)*) – Observations weights for the sample data.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to the objective, gradient, and Hessian-vector functions.

Returns **self** – This object.

Return type obj

predict(*X*, *additional_kwargs={}*)
Make predictions on new data

Note: Using this method requires passing 'pred_fun' in the initialization.

Parameters

- **X** (*array(n_samples, m)*) – New data to pass to user-provided predict function.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to user-provided predict function.

class SQN_free(*mem_size=10*, *bfgs_upd_freq=20*, *min_curvature=0.0001*, *y_reg=None*, *use_grad_diff=False*, *check_nan=True*, *nthreads=-1*, *use_float=False*)

SQN optimizer (free mode)

Optimizes an empirical (convex) loss function over batches of sample data. Compared to class 'SQN', this version lets the user do all the calculations from the outside, only interacting with the object by means of a function that returns a request type and is fed the required calculation through methods 'update_gradient' and 'update_hess_vec'.

Order in which requests are made:

========== loop ========== * calc_grad

... (repeat calc_grad)

if 'use_grad_diff':

- calc_grad_big_batch

else:

- calc_hess_vec

Parameters

- **mem_size** (*int*) – Number of correction pairs to store for approximation of Hessian-vector products.
- **bfgs_upd_freq** (*int*) – Number of iterations (batches) after which to generate a BFGS correction pair.
- **min_curvature** (*float or None*) – Minimum value of s*y / s*s in order to accept a correction pair.
- **y_reg** (*float or None*) – Regularizer for 'y' vector (gets added y_reg * s).
- **use_grad_diff** (*bool*) – Whether to create the correction pairs using differences between gradients instead of Hessian-vector products. These gradients are calculated on a larger batch than the regular ones (given by batch_size * bfgs_upd_freq).
- **check_nan** (*bool*) – Whether to check for variables becoming NaN after each iteration, and reverting the step if they do (will also reset BFGS memory).
- **nthreads** (*int*) – Number of parallel threads to use. If set to -1, will determine the number of available threads and use all of them. Note however that not all the computations can be parallelized.
- **use_float** (*bool*) – Whether to use C 'float' type (np.float32). If 'False' (the default), will use 'double' type (np.float64). The variables and gradient must be of this same dtype.

run_optimizer(*x*, *step_size*)

Continue optimization process after supplying the calculation requested from the last run

Continue the optimization process from where it was left since the last calculation was requested. Will internally do all the updates that are possible until the moment some calculation of function/gradient/hessian-vector is required.

Note: The first time this is run, no calculation needs to be supplied.

Parameters

- **x** (*array(m,)*) – Current values of the variables. Will be modified in-place.
- **step_size** (*float*) – Step size for the next update (note that variables are not updated during all runs).

Returns

request – Dictionary with the calculation required to proceed and iteration information. Structure:

- task : str - one of "calc_grad", "calc_grad_same_batch" (oLBFGS w. 'min_curvature' or 'check_nan'),

"calc_hess_vec" (SQN wo. 'use_grad_diff'), "calc_fun_val_batch" (adaQN w. 'max_incr'), "calc_grad_big_batch" (SQN and adaQN w. 'use_grad_diff'). * requested_on : array(m,) or tuple(array(m,), array(m,)), containing the values on which the request in "task" has to be evaluated. In the case of Hessian-vector products (SQN), the first vector is the values of 'x' and the second is the vector with which the product is required. * info : dict(x_changed_in_run : bool, iteration_number : int, iteration_info : str), iteration_info can be one of "no_problems_encountered", "search_direction_was_nan", "func_increased", "curvature_too_small".

Return type dict

update_gradient(*gradient*)

Pass requested gradient to optimizer

Parameters **gradient** (*array(m,)*) – Gradient calculated as requested, evaluated at values given in "requested_on", calcualted either in a regular batch (task = "calc_grad"), same batch as before (task = "calc_grad_same_batch" - oLBFGS only), or a larger batch of data (task = "calc_grad_big_batch"), perhaps including all the cases from the last such calculation (SQN and adaQN with 'use_grad_diff=True').

update_hess_vec(*hess_vec*)

Pass requested Hessian-vector product to optimizer (task = "calc_hess_vec")

Parameters **hess_vec** (*array(m,)*) – Product of the Hessian evaluated at "requested_on"[0] with the vector "requested_on"[1], calculated a larger batch of data than the gradient, perhaps including all the cases from the last such calculation.

class adaQN(*x0*, *grad_fun*, *obj_fun=None*, *pred_fun=None*, *batches_per_epoch=25*, *step_size=0.1*, *decr_step_size=None*, *shuffle_data=True*, *random_state=1*, *nepochs=25*, *valset_frac=None*, *tol=0.1*, *callback_epoch=None*, *callback_iter=None*, *kwargs_cb={}*, *verbose=True*, *mem_size=10*, *fisher_size=100*, *bfgs_upd_freq=20*, *max_incr=1.01*, *min_curvature=0.0001*, *y_reg=None*, *scal_reg=0.0001*, *rmsprop_weight=None*, *use_grad_diff=False*, *check_nan=True*, *nthreads=-1*, *use_float=False*)

adaQN optimizer

Optimizes an empirical (possibly non-convex) loss function over batches of sample data.

Parameters

- **x0** (*array (m,)*) – Initial values of the variables to optimize (refered hereafter as 'x').
- **grad_fun** (*function(x, X, y, sample_weight, **kwargs) –> array(m,)*) – Function that calculates the empirical gradient at values 'x' on data 'X' and 'y'. Note: output must be one-dimensional and with the same number of entries as 'x', otherwise the Python session might segfault. (The extra keyword arguments are passed in the 'fit' method, not here)
- **obj_fun** (*function(x, X, y, sample_weight, **kwargs) –> float*) – Function that calculates the empirical objective value at values 'x' on data 'X' and 'y'. Will be ignored if passing

'max_incr=None' and no validation set ('valset_frac=None', and no 'valset' passed to fit). (The extra keyword arguments are passed in the 'fit' method, not here)

- **pred_fun** (*None or function(xopt, X)*) – Prediction function taking as input the optimal 'x' values as obtained by the optimization procedure, and new observation 'X' on which to make predictions. If passed, will have an additional method oLBFGS.predict(X, *args) that calls this function with current values of 'x'.
- **batches_per_epoch** (*int*) – Number of batches per epoch (each batch will have the same number of observations except for the last one which might be smaller).
- **step_size** (*float*) – Initial step size to use. (Can be modified after object is already initialized)
- **decr_step_size** (*str "auto", None, or function(initial_step_size, epoch) -> float*) – Function that determines the step size during each epoch, taking as input the initial step size and the epoch number (starting at zero). If "auto", will use 1/sqrt(iteration). If None, will use constant step size. For 'partial_fit', it will take as input the number of iterations of the algorithm rather than epoch, so it's very recommended to provide a custom function when passing data in user-provided batches. Can be modified after the object has been initialized (oLBFGS.decr_step_size)
- **shuffle_data** (*bool*) – Whether to shuffle the data at the beginning of each epoch.
- **random_state** (*int*) – Random seed to use for shuffling data and selecting validation set. The algorithm is deterministic so it's not used for anything else.
- **nepochs** (*int*) – Number of epochs for which to run the optimization procedure. Might terminate earlier if using a validation set for monitoring.
- **valset_frac** (*float(0, 1) or None*) – Percent of the data to use as validation set for early stopping. Can also pass a user-provided validation set to 'fit', in which case it will be ignored. If passing None, will run for the number of epochs passed in 'nepochs'.
- **tol** (*float*) – If the objective function calculated on the validation set decrease by less than 'tol' upon completion of an epoch, will terminate the optimization procedure. Ignored when not using a validation set.
- **callback_epoch** (*None or function*(x, **kwargs)*) – Callback function to call at the end of each epoch
- **callback_iter** (*None or function*(x, **kwargs)*) – Callback function to call at the end of each iteration
- **kwargs_cb** (*tuple*) – Additional arguments to pass to 'callback' and 'stop_crit'. (Can be modified after object is already initialized)
- **verbose** (*bool*) – Whether to print messages when there is some problem during an iteration (e.g. correction pair not meeting minum curvature).
- **mem_size** (*int*) – Number of correction pairs to store for approximation of Hessian-vector products.
- **fisher_size** (*int or None*) – Number of gradients to store for calculation of the empirical Fisher product with gradients. If passing 'None', will force 'use_grad_diff' to 'True'.
- **bfgs_upd_freq** (*int*) – Number of iterations (batches) after which to generate a BFGS correction pair.
- **max_incr** (*float or None*) – Maximum ratio of function values in the validation set under the average values of 'x' during current epoch vs. previous epoch. If the ratio is above this

threshold, the BFGS and Fisher memories will be reset, and 'x' values reverted to their previous average. If not using a validation set, will take a longer batch for function evaluations (same as used for gradients when using 'use_grad_diff=True').

- **min_curvature** (*float or None*) – Minimum value of s*y / s*s in order to accept a correction pair.
- **y_reg** (*float or None*) – regularizer for 'y' vector (gets added y_reg * s)
- **scal_reg** (*float*) – Regularization parameter to use in the denominator for AdaGrad and RMSProp scaling.
- **rmsprop_weight** (*float(0,1) or None*) – If not 'None', will use RMSProp formula instead of AdaGrad for approximated inverse-Hessian initialization. (Recommended to use lower initial step size + passing 'decr_step_size')
- **use_grad_diff** (*bool*) – Whether to create the correction pairs using differences between gradients instead of Fisher matrix. These gradients are calculated on a larger batch than the regular ones (given by batch_size * bfgs_upd_freq). If 'True', fisher_size will be set to None, and empirical Fisher matrix will not be used.
- **check_nan** (*bool*) – Whether to check for variables becoming NaN after each iteration, and reverting the step if they do (will also reset BFGS memory).
- **nthreads** (*int*) – Number of parallel threads to use. If set to -1, will determine the number of available threads and use all of them. Note however that not all the computations can be parallelized.
- **use_float** (*bool*) – Whether to use C 'float' type (np.float32). If 'False' (the default), will use 'double' type (np.float64). The variables and gradient must be of this same dtype.

References

fit(*X*, *y*, *sample_weight=None*, *additional_kwargs={}*, *valset=None*)
Fit model to sample data

Parameters

- **X** (*array(n_samples, m)*) – Sample data to which to fit the model.
- **y** (*array(n_samples,)*) – Labels or target values for the sample data.
- **sample_weight** (*None or array(n_samples,)*) – Observations weights for the sample data.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to the objective, gradient, and Hessian-vector functions.
- **valset** (*tuple(3)*) – User-provided validation set containing (X_val, y_val, sample_weight_val). At the end of each epoch, will calculate objective function on this set, and if the decrease from the objective function in the previous epoch is below tolerance, will terminate procedure earlier. If 'valset_frac' was provided and a validation set is passed, 'valset_frac' will be ignored. Must provide objective function in order to use a validation set.

Returns **self** – This object.

Return type obj

get_x()
Get a copy of current values of the variables

Returns **x** – Current variable values.

Return type array(n,)

niter

partial_fit (*X*, *y*, *sample_weight=None*, *additional_kwargs={}*)
Update model with user-provided batches of data

Note: In SQN and adaQN, the data passed to all calls in partial fit will be stored in a limited-memory container which will be used to calculate Hessian-vector products or large-batch gradients. The size of this container is determined by the inputs 'batch_size' and 'bfgs_upd_freq' passed in the constructor call.

Note: The step size in partial fit is determined by the number of optimizer iterations rather than the number of epochs, thus for a given amount of data, the default step size will be much smaller than when calling 'fit'. Recommended to provide a custom step size function ('decr_step_size' in the initialization), as otherwise the step size sequence will be too small.

Parameters

- **X** (*array(n_samples, m)*) – Sample data to with which to update the model.
- **y** (*array(n_samples,)*) – Labels or target values for the sample data.
- **sample_weight** (*None or array(n_samples,)*) – Observations weights for the sample data.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to the objective, gradient, and Hessian-vector functions.

Returns **self** – This object.

Return type obj

predict (*X*, *additional_kwargs={}*)
Make predictions on new data

Note: Using this method requires passing 'pred_fun' in the initialization.

Parameters

- **X** (*array(n_samples, m)*) – New data to pass to user-provided predict function.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to user-provided predict function.

class adaQN_free (*mem_size=10*, *fisher_size=100*, *bfgs_upd_freq=20*, *max_incr=1.01*, *min_curvature=0.0001*, *scal_reg=0.0001*, *rmsprop_weight=None*, *y_reg=None*, *use_grad_diff=False*, *check_nan=True*, *nthreads=-1*, *use_float=False*)
adaQN optimizer (free mode)

Optimizes an empirical (perhaps non-convex) loss function over batches of sample data. Compared to class 'adaQN', this version lets the user do all the calculations from the outside, only interacting with the object by means of a function that returns a request type and is fed the required calculation through methods 'update_gradient' and 'update_function'.

Order in which requests are made:

========== loop ========== * calc_grad

... (repeat calc_grad)

if max_incr > 0:

- calc_fun_val_batch

if 'use_grad_diff':

- calc_grad_big_batch (skipped if below max_incr)

Parameters

- **mem_size** (*int*) – Number of correction pairs to store for approximation of Hessian-vector products.
- **fisher_size** (*int or None*) – Number of gradients to store for calculation of the empirical Fisher product with gradients. If passing 'None', will force 'use_grad_diff' to 'True'.
- **bfgs_upd_freq** (*int*) – Number of iterations (batches) after which to generate a BFGS correction pair.
- **max_incr** (*float or None*) – Maximum ratio of function values in the validation set under the average values of 'x' during current epoch vs. previous epoch. If the ratio is above this threshold, the BFGS and Fisher memories will be reset, and 'x' values reverted to their previous average. If not using a validation set, will take a longer batch for function evaluations (same as used for gradients when using 'use_grad_diff=True').
- **min_curvature** (*float or None*) – Minimum value of s*y / s*s in order to accept a correction pair.
- **scal_reg** (*float*) – Regularization parameter to use in the denominator for AdaGrad and RMSProp scaling.
- **rmsprop_weight** (*float(0,1) or None*) – If not 'None', will use RMSProp formula instead of AdaGrad for approximated inverse-Hessian initialization.
- **y_reg** (*float or None*) – Regularizer for 'y' vector (gets added y_reg * s).
- **use_grad_diff** (*bool*) – Whether to create the correction pairs using differences between gradients instead of Fisher matrix. These gradients are calculated on a larger batch than the regular ones (given by batch_size * bfgs_upd_freq). If 'True', fisher_size will be set to None, and empirical Fisher matrix will not be used.
- **check_nan** (*bool*) – Whether to check for variables becoming NaN after each iteration, and reverting the step if they do (will also reset BFGS memory).
- **nthreads** (*int*) – Number of parallel threads to use. If set to -1, will determine the number of available threads and use all of them. Note however that not all the computations can be parallelized.
- **use_float** (*bool*) – Whether to use C 'float' type (np.float32). If 'False' (the default), will use 'double' type (np.float64). The variables and gradient must be of this same dtype.

run_optimizer(*x*, *step_size*)

Continue optimization process after supplying the calculation requested from the last run

Continue the optimization process from where it was left since the last calculation was requested. Will internally do all the updates that are possible until the moment some calculation of function/gradient/hessian-vector is required.

Note: The first time this is run, no calculation needs to be supplied.

Parameters

- **x** (*array(m,)*) – Current values of the variables. Will be modified in-place. Do NOT modify the values between runs.
- **step_size** (*float*) – Step size for the next update (note that variables are not updated during all runs).

Returns

request – Dictionary with the calculation required to proceed and iteration information. Structure:

- task : str - one of "calc_grad", "calc_grad_same_batch" (oLBFGS w. 'min_curvature' or 'check_nan'),

"calc_hess_vec" (SQN wo. 'use_grad_diff'), "calc_fun_val_batch" (adaQN w. 'max_incr'), "calc_grad_big_batch" (SQN and adaQN w. 'use_grad_diff'). * requested_on : array(m,) or tuple(array(m,), array(m,)), containing the values on which the request in "task" has to be evaluated. In the case of Hessian-vector products (SQN), the first vector is the values of 'x' and the second is the vector with which the product is required. * info : dict(x_changed_in_run : bool, iteration_number : int, iteration_info : str), iteration_info can be one of "no_problems_encountered", "search_direction_was_nan", "func_increased", "curvature_too_small".

Return type dict

update_function(*fun*)

Pass requested function evaluation to optimizer (task = "calc_fun_val_batch")

Parameters **fun** (*float*) – Function evaluated at "requested_on" under a validation set or a larger batch, perhaps including all the cases from the last such calculation.

update_gradient(*gradient*)

Pass requested gradient to optimizer

Parameters **gradient** (*array(m,)*) – Gradient calculated as requested, evaluated at values given in "requested_on", calcualted either in a regular batch (task = "calc_grad"), same batch as before (task = "calc_grad_same_batch" - oLBFGS only), or a larger batch of data (task = "calc_grad_big_batch"), perhaps including all the cases from the last such calculation (SQN and adaQN with 'use_grad_diff=True').

class oLBFGS(*x0*, *grad_fun*, *obj_fun=None*, *pred_fun=None*, *batches_per_epoch=25*, *step_size=0.001*, *decr_step_size='auto'*, *shuffle_data=True*, *random_state=1*, *nepochs=25*, *valset_frac=None*, *tol=0.1*, *callback_epoch=None*, *callback_iter=None*, *kwargs_cb={}*, *verbose=True*, *mem_size=10*, *hess_init=None*, *min_curvature=0.0001*, *y_reg=None*, *check_nan=True*, *nthreads=-1*, *use_float=False*)

oLBFGS optimizer

Optimizes an empirical (convex) loss function over batches of sample data.

Parameters

- **x0** (*array (m,)*) – Initial values of the variables to optimize (refered hereafter as 'x').
- **grad_fun** (*function(x, X, y, sample_weight, **kwargs) –> array(m,)*) – Function that calculates the empirical gradient at values 'x' on data 'X' and 'y'. Note: output must be

one-dimensional and with the same number of entries as 'x', otherwise the Python session might segfault. (The extra keyword arguments are passed in the 'fit' method, not here)

- **obj_fun** (*function(x, X, y, sample_weight, **kwargs) –> float*) – Function that calculates the empirical objective value at values 'x' on data 'X' and 'y'. Only used when using a validation set ('valset_frac' not None, or 'valset' passed to fit). Ignored when fitting the data in user-provided batches. (The extra keyword arguments are passed in the 'fit' method, not here)
- **pred_fun** (*None or function(xopt, X)*) – Prediction function taking as input the optimal 'x' values as obtained by the optimization procedure, and new observation 'X' on which to make predictions. If passed, will have an additional method oLBFGS.predict(X, *args) that calls this function with current values of 'x'.
- **batches_per_epoch** (*int*) – Number of batches per epoch (each batch will have the same number of observations except for the last one which might be smaller).
- **step_size** (*float*) – Initial step size to use. (Can be modified after object is already initialized)
- **decr_step_size** (*str "auto", None, or function(initial_step_size, epoch) -> float*) – Function that determines the step size during each epoch, taking as input the initial step size and the epoch number (starting at zero). If "auto", will use 1/sqrt(iteration). If None, will use constant step size. For 'partial_fit', it will take as input the number of iterations of the algorithm rather than epoch, so it's very recommended to provide a custom function when passing data in user-provided batches. Can be modified after the object has been initialized (oLBFGS.decr_step_size)
- **shuffle_data** (*bool*) – Whether to shuffle the data at the beginning of each epoch.
- **random_state** (*int*) – Random seed to use for shuffling data and selecting validation set. The algorithm is deterministic so it's not used for anything else.
- **nepochs** (*int*) – Number of epochs for which to run the optimization procedure. Might terminate earlier if using a validation set for monitoring.
- **valset_frac** (*float(0, 1) or None*) – Percent of the data to use as validation set for early stopping. Can also pass a user-provided validation set to 'fit', in which case it will be ignored. If passing None, will run for the number of epochs passed in 'nepochs'.
- **tol** (*float*) – If the objective function calculated on the validation set decrease by less than 'tol' upon completion of an epoch, will terminate the optimization procedure. Ignored when not using a validation set.
- **callback_epoch** (*None or function*(x, **kwargs)*) – Callback function to call at the end of each epoch
- **callback_iter** (*None or function*(x, **kwargs)*) – Callback function to call at the end of each iteration
- **kwargs_cb** (*tuple*) – Additional arguments to pass to 'callback' and 'stop_crit'. (Can be modified after object is already initialized)
- **verbose** (*bool*) – Whether to print messages when there is some problem during an iteration (e.g. correction pair not meeting minum curvature).
- **mem_size** (*int*) – Number of correction pairs to store for approximation of Hessian-vector products.
- **hess_init** (*float or None*) – value to which to initialize the diagonal of H0. If passing 0, will use the same initializion as for SQN (s_last*y_last / y_last*y_last).

- **min_curvature** (*float or None*) – Minimum value of s*y / s*s in order to accept a correction pair.
- **y_reg** (*float or None*) – regularizer for 'y' vector (gets added y_reg * s)
- **check_nan** (*bool*) – Whether to check for variables becoming NaN after each iteration, and reverting the step if they do (will also reset BFGS memory).
- **nthreads** (*int*) – Number of parallel threads to use. If set to -1, will determine the number of available threads and use all of them. Note however that not all the computations can be parallelized.
- **use_float** (*bool*) – Whether to use C 'float' type (np.float32). If 'False' (the default), will use 'double' type (np.float64). The variables and gradient must be of this same dtype.

References

fit (*X*, *y*, *sample_weight=None*, *additional_kwargs={}*, *valset=None*)
Fit model to sample data

Parameters

- **X** (*array(n_samples, m)*) – Sample data to which to fit the model.
- **y** (*array(n_samples,)*) – Labels or target values for the sample data.
- **sample_weight** (*None or array(n_samples,)*) – Observations weights for the sample data.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to the objective, gradient, and Hessian-vector functions.
- **valset** (*tuple(3)*) – User-provided validation set containing (X_val, y_val, sample_weight_val). At the end of each epoch, will calculate objective function on this set, and if the decrease from the objective function in the previous epoch is below tolerance, will terminate procedure earlier. If 'valset_frac' was provided and a validation set is passed, 'valset_frac' will be ignored. Must provide objective function in order to use a validation set.

Returns **self** – This object.

Return type obj

get_x ()
Get a copy of current values of the variables

Returns **x** – Current variable values.

Return type array(n,)

niter

partial_fit (*X*, *y*, *sample_weight=None*, *additional_kwargs={}*)
Update model with user-provided batches of data

Note: In SQN and adaQN, the data passed to all calls in partial fit will be stored in a limited-memory container which will be used to calculate Hessian-vector products or large-batch gradients. The size of this container is determined by the inputs 'batch_size' and 'bfgs_upd_freq' passed in the constructor call.

Note: The step size in partial fit is determined by the number of optimizer iterations rather than the number of epochs, thus for a given amount of data, the default step size will be much smaller than when

calling 'fit'. Recommended to provide a custom step size function ('decr_step_size' in the initialization), as otherwise the step size sequence will be too small.

Parameters

- **X** (*array(n_samples, m)*) – Sample data to with which to update the model.
- **y** (*array(n_samples,)*) – Labels or target values for the sample data.
- **sample_weight** (*None or array(n_samples,)*) – Observations weights for the sample data.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to the objective, gradient, and Hessian-vector functions.

Returns **self** – This object.

Return type obj

predict (*X*, *additional_kwargs={}*)
Make predictions on new data

Note: Using this method requires passing 'pred_fun' in the initialization.

Parameters

- **X** (*array(n_samples, m)*) – New data to pass to user-provided predict function.
- **additional_kwargs** (*dict*) – Additional keyword arguments to pass to user-provided predict function.

class oLBFGS_free (*mem_size=10*, *hess_init=None*, *min_curvature=0.0001*, *y_reg=None*, *check_nan=True*, *nthreads=-1*, *use_float=False*)
oLBFGS optimizer (free mode)

Optimizes an empirical (convex) loss function over batches of sample data. Compared to class 'oLBFGS', this version lets the user do all the calculations from the outside, only interacting with the object by means of a function that returns a request type and is fed the required calculation through a method 'update_gradient'.

Order in which requests are made:

========== loop ========== * calc_grad * calc_grad_same_batch (might skip if using check_nan) ==========

Parameters

- **mem_size** (*int*) – Number of correction pairs to store for approximation of Hessian-vector products.
- **hess_init** (*float or None*) – value to which to initialize the diagonal of H0. If passing 'None', will use the same initializion as for SQN (s_last*y_last / y_last*y_last).
- **min_curvature** (*float or None*) – Minimum value of s*y / s*s in order to accept a correction pair.
- **y_reg** (*float or None*) – Regularizer for 'y' vector (gets added y_reg * s).
- **check_nan** (*bool*) – Whether to check for variables becoming NaN after each iteration, and reverting the step if they do (will also reset BFGS memory).

- **nthreads** (*int*) – Number of parallel threads to use. If set to -1, will determine the number of available threads and use all of them. Note however that not all the computations can be parallelized.
- **use_float** (*bool*) – Whether to use C 'float' type (np.float32). If 'False' (the default), will use 'double' type (np.float64). The variables and gradient must be of this same dtype.

run_optimizer(*x*, *step_size*)

Continue optimization process after supplying the calculation requested from the last run

Continue the optimization process from where it was left since the last calculation was requested. Will internally do all the updates that are possible until the moment some calculation of function/gradient/hessian-vector is required.

Note: The first time this is run, no calculation needs to be supplied.

Parameters

- **x** (*array(m,)*) – Current values of the variables. Will be modified in-place. Do NOT modify the values between runs.
- **step_size** (*float*) – Step size for the next update (note that variables are not updated during all runs).

Returns

request – Dictionary with the calculation required to proceed and iteration information. Structure:

- task : str - one of "calc_grad", "calc_grad_same_batch" (oLBFGS w. 'min_curvature' or 'check_nan'),

"calc_hess_vec" (SQN wo. 'use_grad_diff'), "calc_fun_val_batch" (adaQN w. 'max_incr'), "calc_grad_big_batch" (SQN and adaQN w. 'use_grad_diff'). * requested_on : array(m,) or tuple(array(m,), array(m,)), containing the values on which the request in "task" has to be evaluated. In the case of Hessian-vector products (SQN), the first vector is the values of 'x' and the second is the vector with which the product is required. * info : dict(x_changed_in_run : bool, iteration_number : int, iteration_info : str), iteration_info can be one of "no_problems_encountered", "search_direction_was_nan", "func_increased", "curvature_too_small".

Return type dict

update_gradient(*gradient*)

Pass requested gradient to optimizer

Parameters **gradient** (*array(m,)*) – Gradient calculated as requested, evaluated at values given in "requested_on", calcualted either in a regular batch (task = "calc_grad"), same batch as before (task = "calc_grad_same_batch" - oLBFGS only), or a larger batch of data (task = "calc_grad_big_batch"), perhaps including all the cases from the last such calculation (SQN and adaQN with 'use_grad_diff=True').

class StochasticLogisticRegression(*reg_param=0.001*, *fit_intercept=True*, *random_state=1*, *optimizer='SQN'*, *step_size=0.1*, *valset_frac=0.1*, *verbose=False*, ***optimizer_kwargs*)

Logistic Regression fit with stochastic quasi-Newton optimizer

Parameters

- **reg_param** (*float*) – Strength of l2 regularization. Note that the loss function has an average log-loss over observations, so the optimal regulatization will likely be a lot smaller than for scikit-learn's (which uses sum instead).
- **step_size** (*float*) – Initial step size to use. Note that it will be decreased after each epoch when using 'fit', but will not be decreased after calling 'partial_fit'.
- **fit_intercept** (*bool*) – Whether to add an intercept to the model parameters.
- **random_state** (*int*) – Random seed to use.
- **optimizer** (*str, one of 'oLBFGS', 'SQN', 'adaQN'*) – Optimizer to use.
- **optimizer_kwargs** (*dict, optional*) – Additional options to pass to the optimizer (see each optimizer's documentation).

coef_

fit (*X*, *y*, *sample_weight=None*)
Fit Logistic Regression model in stochastic batches

Parameters

- **X** (*array(n_samples, n_features)*) – Covariates (features).
- **y** (*array(n_samples,) or array(n_samples, n_classes)*) – Labels for each observation (must be already one-hot encoded).
- **sample_weight** (*array(n_samples,) or None*) – Observation weights for each data point.

Returns **self** – This object

Return type obj

intercept_

partial_fit (*X*, *y*, *sample_weight=None*, *classes=None*, *decr_step_size=False*)
Fit Logistic Regression model in stochastic batches

Parameters

- **X** (*array(n_samples, n_features)*) – Covariates (features).
- **y** (*array(n_samples,) or array(n_samples, n_classes)*) – Labels for each observation (must be already one-hot encoded).
- **sample_weight** (*array(n_samples,) or None*) – Observation weights for each data point.
- **classes** (*None*) – Not used. Kept there for compatibility with other packages that assume scikit-learn's API.
- **decr_step_size** (*bool*) – Whether to decrease or not decrease the step size after the update is done, according to the function 'decr_step_size' passed at initialization.

Returns **self** – This object

Return type obj

predict (*X*)
Predict the class of new observations

Parameters **X** (*array(n_samples, n_features)*) – Input data on which to predict classes.

Returns **pred** – Predicted class for each observation

Return type array(n_samples,)

predict_proba(*X*)

Predict class probabilities for new observations

Parameters **X** (*array(n_samples, n_features)*) – Input data on which to predict class probabilities.

Returns **pred** – Predicted class probabilities for each observation

Return type array(n_samples, n_classes)

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index