

# Stochastic Gradient Descent

Marc Berghouse and Lazaro Perez

# What is Stochastic Gradient Descent (SGD)

- Gradient descent
  - An algorithm used to determine the minimum of any differentiable function
  - The gradient is a multivariable derivative, and it gives the direction of the steepest ascent
    - In gradient descent we use the negative of the gradient to find the direction of steepest descent
- Stochastic means random sampling is involved
  - Standard gradient descent uses all data points to update the weights on each iteration
  - Stochastic gradient descent only uses one random data point for each iteration
- **SGD is an optimization technique** and doesn't correspond to a particular model
  - It can be considered one of many methods for training a variety of machine learning models
  - Basically just a way to reach the minimum of your loss function
- A numerical approximation to the classic method of “take the derivative and set it equal to 0”
  - For easy functions we can solve this classically, but for complex functions without analytical solutions for roots, we need to use numerical methods to approximate

# Common Loss Functions

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad \text{MAE} = \frac{1}{n} \sum_{i=1}^n |Y_i - \hat{Y}_i|$$

## ● Regression

- Mean square error (MSE) AKA L2 loss
- Mean absolute error (MAE) AKA L1 loss
- Mean square log error
- Huber (L1+L2)

■ Low outlier penalty

- Log-cosh

■ Low outlier penalty

## ● Classification

- Cross-entropy

■ Binary

■ Categorical

- Hinge

### Binary cross entropy

$$L_{BCE} = -\frac{1}{n} \sum_{i=1}^n (Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \cdot \log (1 - \hat{Y}_i))$$

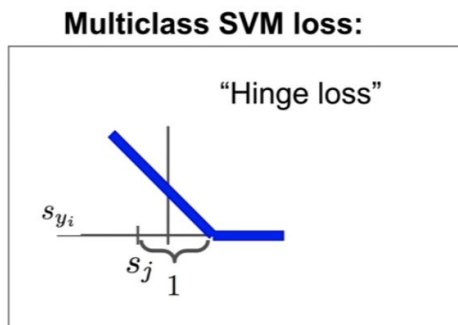
Quadratic

$$L_{\delta} = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

Linear

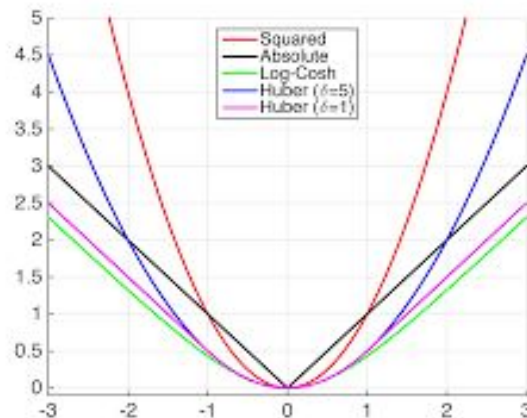
Huber

$$\text{logcosh} = \frac{1}{n} \sum_{i=1}^n \log(\cosh(\hat{y}_i - y_i))$$

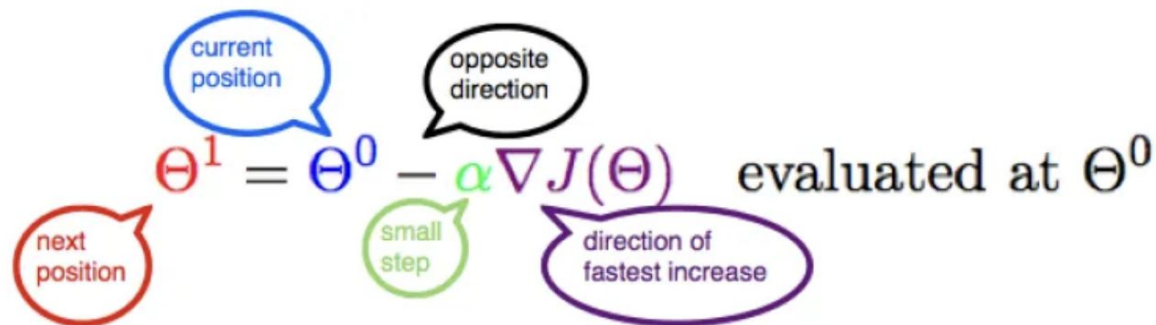


$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$



# How Does Gradient Descent Work?



The diagram illustrates the gradient descent formula:  $\Theta^1 = \Theta^0 - \alpha \nabla J(\Theta)$  evaluated at  $\Theta^0$ . Callouts explain the components:  $\Theta^0$  is the 'current position',  $\Theta^1$  is the 'next position',  $\alpha$  is a 'small step', and  $\nabla J(\Theta)$  is the 'direction of fastest increase' (noted as 'opposite direction' in the callout).

- Iteratively solve for the minimum of a loss function
  - Here the loss function is represented by  $J$
- We calculate the gradient (multivariable derivative) of the cost (loss) function to get the direction of the fastest increase in slope
  - The negative means we calculate the direction of fastest decrease of the cost function
- Theta represents the parameters of the model
- Alpha represents the learning rate

<https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e>

# Math Behind SGD

$$w - \frac{\eta}{n} \sum_{i=1}^n \nabla Q_i(w), \quad w := w - \eta \nabla Q_i(w).$$

Suppose we want to fit a straight line  $\hat{y} = w_1 + w_2 x$  to a training set with observations  $(x_1, x_2, \dots, x_n)$  and corresponding estimated responses  $(\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n)$  using [least squares](#). The objective function to be minimized is:

$$Q(w) = \sum_{i=1}^n Q_i(w) = \sum_{i=1}^n (\hat{y}_i - y_i)^2 = \sum_{i=1}^n (w_1 + w_2 x_i - y_i)^2.$$

The last line in the above pseudocode for this specific problem will become:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial}{\partial w_1} (w_1 + w_2 x_i - y_i)^2 \\ \frac{\partial}{\partial w_2} (w_1 + w_2 x_i - y_i)^2 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \eta \begin{bmatrix} 2(w_1 + w_2 x_i - y_i) \\ 2x_i(w_1 + w_2 x_i - y_i) \end{bmatrix}.$$

- $W$  corresponds to the parameters of the model (for linear regression in this case)

# Simple Example

$$w := w - \eta \nabla Q_i(w).$$

- Say our loss function is  $y=x^2$ , which corresponds to 1D MSE
  - To approximate the minimum, we make an initial guess of a value, say  $x=1$
  - We then use the algorithm to calculate our next value of  $x$  in the iteration
  - Let's assume a learning rate of .2
  - All we need is the value of  $x$  and the gradient of  $y$  ( $2x$  in this case)
  - $x_1 = x_0 - .2 * 2x_0 = .8$
  - $x_2 = x_1 - .2 * 2x_1 = .64$
  - $x_3 = x_2 - .2 * 2x_2 = .512$

# Multivariable Example

The general form of the gradient vector is given by:

$$\nabla f(x,y) = 2xi + 4yj$$

- Say  $f(x,y) = x^2 + 2y^2$

Two iterations of the algorithm,  $T=2$  and  $\eta=0.1$  are shown below

1. Initial  $t=0$

- $x[0] = (4,3)$  # This is just a randomly chosen point

2. At  $t = 1$

- $x[1] = x[0] - \eta \nabla f(x[0])$
- $x[1] = (4,3) - 0.1*(8,12)$
- $x[1] = (3.2,1.8)$

3. At  $t=2$

- $x[2] = x[1] - \eta \nabla f(x[1])$
- $x[2] = (3.2,1.8) - 0.1*(6.4,7.2)$
- $x[2] = (2.56,1.08)$

<https://machinelearningmastery.com/a-gentle-introduction-to-gradient-descent-procedure/>

## Adding Momentum

<https://machinelearningmastery.com/a-gentle-introduction-to-gradient-descent-procedure/>

Gradient descent can run into problems such as:

1. Oscillate between two or more points
2. Get trapped in a local minimum
3. Overshoot and miss the minimum point

To take care of the above problems, a momentum term can be added to the update equation of gradient descent algorithm as:

$$x[t] = x[t-1] - \eta \nabla f(x[t-1]) + \alpha \Delta x[t-1]$$

where  $\Delta x[t-1]$  represents the change in  $x$ , i.e.,

$$\Delta x[t] = x[t] - x[t-1]$$

# Momentum and Gradient Ascent

## About Gradient Ascent

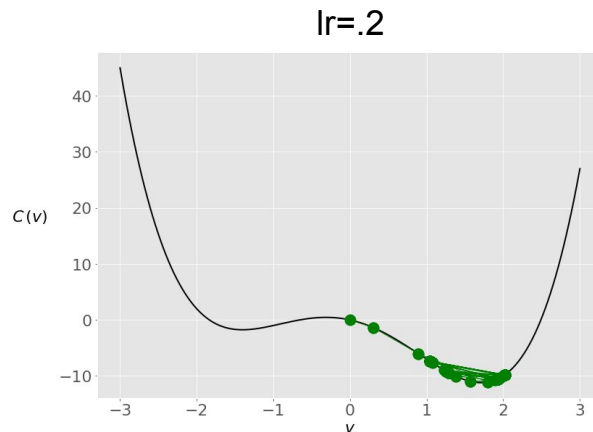
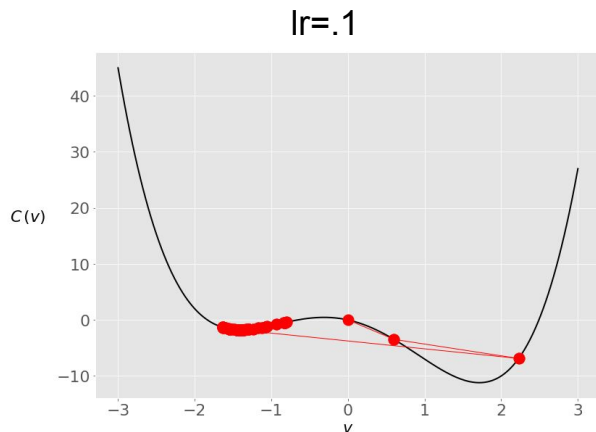
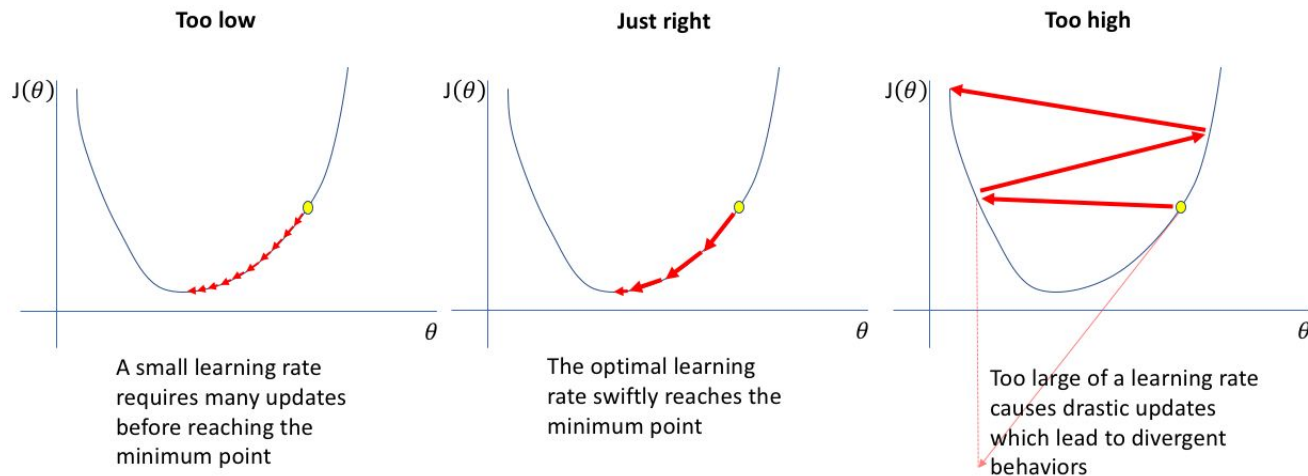
There is a related gradient ascent procedure, which finds the maximum of a function. In gradient descent we follow the direction of the rate of maximum decrease of a function. It is the direction of the negative gradient vector. Whereas, in gradient ascent we follow the direction of maximum rate of increase of a function, which is the direction pointed to by the positive gradient vector. We can also write a maximization problem in terms of a maximization problem by adding a negative sign to  $f(x)$ , i.e.,

1 maximize $f(x)$ w.r.t $x$	is equivalent to	minimize $-f(x)$ w.r.t $x$
-----------------------------	------------------	----------------------------



# Impact of Learning Rate

<https://www.jeremyjordan.me/nn-learning-rate/>



- For real data the loss function will have lots of different minima
- Most of deep learning and some machine learning is about finding the global minima of the non-convex loss function

# Where is SGD Used?

- Any algorithm that optimizes a loss function to make predictions
  - SVM
  - Logistic and linear regression
  - Multi-layer perceptrons (neural networks)
    - One of the most popular optimization algorithms in deep learning
  - K-means clustering
  - Lots more
- Gradient boosting is not exactly SGD, but it is very similar
  - The general idea with boosting is to make a strong predictor from an ensemble of weak predictors (like decision trees)
  - After calculating the loss of a tree, we include that residual in the loss function of our next tree
  - Gradient descent uses updated parameters to descend the gradient
  - Gradient boosting uses new models to descend the gradient

# SGD vs Batch Gradient Descent

## Stochastic Gradient Descent

### Pros:

- ✓ Computes faster since it goes through one example at a time
- ✓ The randomization helps to avoid cycles and repeat of examples
- ✓ Lesser computation burden which allows for lower standard the error
- ✓ Because the example size is less than the training set, there tends to be more noise which allows for the improved generalization error

### Cons:

- ✓ It is usually noisier and this can result in a longer run time
- ✓ Results in larger variance because it works with one example per iteration

## Gradient Descent (Batch Gradient Descent)

### Pros:

- ✓ The trajectory towards the global minimum is always straightforward and it is always guaranteed to converge
- ✓ Even while the learning process is ongoing, the learning rate can be fixed to allow improvements
- ✓ It produces no noise and gives a lower standard error
- ✓ It produces an unbiased estimate of the gradients

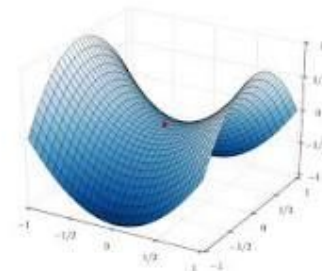
### Cons:

- ✓ It is slower and takes too much time
- ✓ It is computationally expensive with a very high computing burden

<https://sdsclub.com/stochastic-gradient-descent-vs-gradient-descent-a-head-to-head-comparison/>

# Alternatives to SGD

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$



- Machine Learning

- Newton's method

- Root-finding algorithm
    - To find the minimum we must set the 1st derivative equal to 0, which means Newton's method will then require the 2nd derivative
    - Slower due to costs of calculating 2nd derivative
    - May get stuck at local minima or saddle points (gradients are 0, but not a local minimum)
    - If 2nd derivative is 0, we get an error (division by 0)
    - Despite the problems, general consensus is Newton boosting will outperform gradient boosting
      - Newton boosting is used in XGBoost, which is commonly considered one of the best ML models

- L-BFGS

- Quasi-Newton method that estimates the 2nd derivative (AKA the Hessian)
    - Attempts to retain the accuracy of Newton's method with lower computational costs

- Levenberg–Marquardt algorithm

- Faster convergence than SGD, but computational expense is huge  $O(n^3)$
    - No guarantee convergence will occur at a lower minima than with SGD

- Deep Learning

- AdaGrad
  - RMSprop
  - Adam
  - Lots more

<https://stats.stackexchange.com/questions/253632/why-is-newtons-method-not-widely-used-in-machine-learning>

# Applications of SGD in Hydrology

- No specific research on the benefits of SGD over other optimizers in hydrology
- SGD can be used for just about any deep learning method
- SGD can be used for most ML methods that use optimization to find the minimum of the loss function
- Most commonly found in deep learning applications
- Samadianfard et al (2022) show that a random forest model optimized with SGD outperforms a standard random forest model for suspended sediment prediction: <https://link.springer.com/article/10.1007/s00521-021-06550-1>
- Huang et al (2022) show that logistic regression with SGD optimization outperforms bayesian network, decision table, and radial basis function network (RBFN) models for landslide susceptibility prediction: <https://www.mdpi.com/2073-445X/11/3/436>

# Applications of SGD in Hydrology

- Hongwei Li et al (2021) show that a neural net with SGD optimization outperforms a neural net with particle swarm optimization for the prediction of longitudinal dispersion coefficients in streams:

<https://www.tandfonline.com/doi/full/10.1080/19942060.2022.2141896>

- Mahsa et al (2018) show that SGD performs worse than all other common deep learning optimizers for daily river runoff prediction:

<https://www.j-kosham.or.kr/journal/view.php?doi=10.9798/KOSHAM.2018.18.6.377>

Table 1

Average of Performance Measurement as RMSE, MAE, and Bias for Validation Dataset with Different Algorithms in Neural Network as well as the Number of Epoch at the Last Column

	RMSE	MAE	Bias	Num Epoch
SGD	389.4	194.9	-40.8	254.0
Adagrade	380.8	190.8	-39.3	304.5
RMSprop	377.4	188.0	-40.8	249.9
AdaDelta	381.1	188.0	-41.9	222.3
Adam	369.6	185.5	-37.9	246.1
Nadam	373.4	186.3	-37.5	192.2

# Applications of SGD in Hydrology

- Gradient boosting is often used to improve predictive models in hydrology
  - [Rapid Prediction Model for Urban Floods Based on a Light Gradient Boosting Machine Approach and Hydrological–Hydraulic Model | SpringerLink](#)
  - [Streamflow forecasting using extreme gradient boosting model coupled with Gaussian mixture model - ScienceDirect](#)
  - Regionalization of hydrological model parameters using gradient boosting machine:  
<https://hess.copernicus.org/articles/26/505/2022/>
  - [Ensemble machine learning paradigms in hydrology: A review - ScienceDirect](#)
  - [Dynamic Streamflow Simulation via Online Gradient-Boosted Regression Tree | Journal of Hydrologic Engineering | Vol 24, No 10](#)
  - Application of extreme gradient boosting and parallel random forest algorithms for assessing groundwater spring potential using DEM-derived factors:  
<https://www.sciencedirect.com/science/article/pii/S0022169420306570>
  - <https://iwaponline.com/ws/article/21/2/668/78483/Remotely-observed-variations-of-reservoir-load>

# SGD Classification in SKlearn

## `sklearn.linear_model.SGDClassifier`

```
class sklearn.linear_model.SGDClassifier(loss='hinge', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True,
max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, n_jobs=None, random_state=None, learning_rate='optimal',
eta0=0.0, power_t=0.5, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, class_weight=None,
warm_start=False, average=False) ¶
```

[\[source\]](#)

**Parameters:** **loss** : {'hinge', 'log\_loss', 'log', 'modified\_huber', 'squared\_hinge', 'perceptron', 'squared\_error', 'huber', 'epsilon\_insensitive', 'squared\_epsilon\_insensitive'}, default='hinge'


The loss function to be used.

- 'hinge' gives a linear SVM.
- 'log\_loss' gives logistic regression, a probabilistic classifier.
- **'modified\_huber' is another smooth loss that brings tolerance to outliers as well as probability estimates.**
- 'squared\_hinge' is like hinge but is quadratically penalized.
- 'perceptron' is the linear loss used by the perceptron algorithm.
- The other losses, 'squared\_error', 'huber', 'epsilon\_insensitive' and 'squared\_epsilon\_insensitive' are designed for regression but can be useful in classification as well; see [SGDRegressor](#) for a description.



# SGD Regression in SKlearn

- Note that the choice of loss influences the model choice
- SGD in SKlearn is normally only used for large amounts of data, since it is often faster, but less accurate, than the default algorithm for a given model

```
class sklearn.linear_model.SGDRegressor(loss='squared_error', *, penalty='l2', alpha=0.0001, l1_ratio=0.15,
fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, random_state=None,
learning_rate='invscaling', eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5,
warm_start=False, average=False) 
```

[\[source\]](#)

## **penalty : {'l2', 'l1', 'elasticnet', None}, default='l2'**

The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2'. No penalty is added when set to `None`.

## **alpha : float, default=0.0001**

Constant that multiplies the regularization term. The higher the value, the stronger the regularization. Also used to compute the learning rate when set to `learning_rate` is set to 'optimal'.

## **l1\_ratio : float, default=0.15**

The Elastic Net mixing parameter, with  $0 \leq \text{l1\_ratio} \leq 1$ . `l1_ratio=0` corresponds to L2 penalty, `l1_ratio=1` to L1. Only used if `penalty` is 'elasticnet'.

## **fit\_intercept : bool, default=True**

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

## **max\_iter : int, default=1000**

The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the `partial_fit` method.

*New in version 0.19.*

## **tol : float or None, default=1e-3**

The stopping criterion. If it is not None, training will stop when  $(\text{loss} > \text{best\_loss} - \text{tol})$  for `n_iter_no_change` consecutive epochs. Convergence is checked against the training loss or the validation loss depending on the `early_stopping` parameter.

## **loss : str, default='squared\_error'**

The loss function to be used. The possible values are 'squared\_error', 'huber', 'epsilon\_insensitive', or 'squared\_epsilon\_insensitive'.

The 'squared\_error' refers to the ordinary least squares fit. 'huber' modifies 'squared\_error' to focus less on getting outliers correct by switching from squared to linear loss past a distance of epsilon. 'epsilon\_insensitive' ignores errors less than epsilon and is linear past that; this is the loss function used in SVR. 'squared\_epsilon\_insensitive' is the same but becomes squared loss past a tolerance of epsilon.



# Gradient Boosting in SKlearn (Gradient Boosting Applied to Decision Trees)

## `sklearn.ensemble.GradientBoostingRegressor` ¶

```
class sklearn.ensemble.GradientBoostingRegressor(*, loss='squared_error', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, alpha=0.9, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)
```

[\[source\]](#)

## `sklearn.ensemble.GradientBoostingClassifier`

```
class sklearn.ensemble.GradientBoostingClassifier(*, loss='log_loss', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0) ¶
```

# Conclusions About SGD in Hydrology

- Can be an effective way to improve the performance of standard optimization methods for a variety of machine learning models
- Lots of tuning parameters means you have a huge parameter/feature space to search for good results
  - The good results are out there, but they may be hard to find
- Often used as an optimization method for deep learning, although it rarely generates the best results