



The objective of this lab is for you to program in Matlab gradient descent (GD) and stochastic gradient descent (SGD) for two linear models (for regression and for classification), apply them to some datasets and observe their behavior. The TA will first demonstrate the results of the algorithms on several datasets, and then you will program them, replicate those results, and further explore the datasets with the algorithms. You can use the textbook, lecture notes and your own notes.

Important: develop your code so it works with inputs $\mathbf{x} \in \mathbb{R}^D$ for any dimension D (and, for regression, with outputs $\mathbf{y} \in \mathbb{R}^{D_y}$ for any dimension D_y). It is as easy as for dimension 1 if you use vectorized code in Matlab, and it should look very similar to the actual equations. Start with the easiest case, which is linear regression, on a toy dataset where $x_n, y_n \in \mathbb{R}$. Implement GD and SGD there, make sure they work as expected and understand their behavior as a function of their parameters (learning rate and minibatch size). Then move on to logistic regression, which is a bit more complicated, and to higher-dimensional datasets.

I Datasets

Firstly, construct your own toy datasets to visualize the result easily and be able to get the algorithm right. Take the input instances $\{\mathbf{x}_n\}_{n=1}^N$ in \mathbb{R} or \mathbb{R}^2 and the labels $\{y_n\}_{n=1}^N$ in \mathbb{R} (regression) or $\{0, 1\}$ (classification).

Then, try the MNIST dataset of handwritten digits, with instances $\mathbf{x} \in \mathbb{R}^D$ (where $D = 784$). For classification with logistic regression, use the labels $y_n \in \{0, \dots, 9\}$ (for binary classification, pick instances from only two digit classes and use labels $y_n \in \{0, 1\}$). For linear regression, create a ground-truth mapping $\mathbf{y} = \mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$ as follows:

- A random mapping with output dimension D_y , e.g. take $\mathbf{A}_{D_y \times D}$ and $\mathbf{b}_{D_y \times 1}$ with elements in $\mathcal{N}(0, 1)$.
- A mapping that rotates, scales, shifts and possibly clips the input image \mathbf{x} and adds noise to it, e.g.  \rightarrow . This makes it easy to visualize the result, since the desired output $\mathbf{f}(\mathbf{x})$ for an image \mathbf{x} should look like \mathbf{x} but transformed accordingly.

II (Stochastic) gradient descent for linear regression

Given a sample $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$ with $\mathbf{x}_n \in \mathbb{R}^D$ and $y_n \in \mathbb{R}$, we solve a linear least-squares regression problem by minimizing

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N (y_n - \mathbf{w}^T \mathbf{x}_n)^2 = \sum_{n=1}^N e(\mathbf{w}; \mathbf{x}_n, y_n) \quad \text{with} \quad e(\mathbf{w}; \mathbf{x}_n, y_n) = \frac{1}{2} (y_n - \mathbf{w}^T \mathbf{x}_n)^2$$

where we assume the last element of each \mathbf{x}_n vector is equal to 1, so that w_D is the bias parameter (this simplifies the notation). $E(\mathbf{w})$ is a quadratic objective function over \mathbf{w} with a unique minimizer that can be found in closed form by solving the normal equations (obtained by computing the gradient of E wrt \mathbf{w} , setting it to zero and solving for \mathbf{w}):

$\mathbf{X}\mathbf{X}^T \mathbf{w}^* = \mathbf{X}\mathbf{y} \Leftrightarrow \mathbf{w}^* = (\mathbf{X}\mathbf{X}^T)^{-1} \mathbf{X}\mathbf{y}$, where $\mathbf{X}_{D \times N} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, $\mathbf{w}_{D \times 1}^* = (w_1, \dots, w_D)^T$, $\mathbf{y}_{N \times 1} = (y_1, \dots, y_N)^T$.

This can be done with a routine for solving linear systems, e.g. in Matlab, use `linsolve` or the “\” operator. You can also use `inv` but this is numerically more costly and less stable.

The minimizer \mathbf{w}^* of E can also be found iteratively by gradient descent (GD) and stochastic gradient descent (SGD), starting from an initial \mathbf{w} . These use the following updates (obtained by computing the gradient $\nabla E(\mathbf{w})$ of E wrt \mathbf{w}):

$$\text{GD: } \Delta \mathbf{w} = \eta \sum_{n=1}^N (y_n - \mathbf{w}^T \mathbf{x}_n) \mathbf{x}_n \quad \text{SGD: } \Delta \mathbf{w} = \eta \sum_{n \in \mathcal{B}} (y_n - \mathbf{w}^T \mathbf{x}_n) \mathbf{x}_n$$

where η is the learning rate (step size) and \mathcal{B} is a minibatch (i.e., a subset of $1 < |\mathcal{B}| < N$ points). For GD, at each iteration we set $\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$ with $\Delta \mathbf{w} = -\eta \nabla E(\mathbf{w})$. For SGD, after each minibatch we set $\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$ with $\Delta \mathbf{w} = -\eta \sum_{n \in \mathcal{B}} \nabla e(\mathbf{w}; \mathbf{x}_n)$, and repeat over all minibatches to complete one iteration (one epoch), which passes over the whole data once. If the minibatches contain a single data point ($|\mathcal{B}| = 1$), then there are N minibatches and we do pure online learning. If there is a single minibatch containing all points ($|\mathcal{B}| = N$), we do pure batch learning (identical to GD). The fastest training occurs for an intermediate (usually small) minibatch size. It also helps to reorder at random the data points at the beginning of each epoch (so the minibatches vary, and are scanned in a different order), rather than using the same, original order at every epoch.

What you have to do: toy problem

- Firstly, verify that the GD/SGD updates above are correct, by obtaining the gradient of $E(\mathbf{w})$ with pen and paper. Then, implement GD and SGD by programming the updates with a “for” loop. Run them for 100 iterations $\mathbf{w}^0, \mathbf{w}^{(1)}, \dots, \mathbf{w}^{(100)}$ from an initial $\mathbf{w} = \mathbf{w}^{(0)}$ (equal to small random numbers, e.g. uniform in $[-0.01, 0.01]$).
- Does the error $E(\mathbf{w})$ for GD and for SGD approach the optimal error $E(\mathbf{w}^*)$ (where \mathbf{w}^* is the solution to the normal equations)? How fast does it approach it? Does $E(\mathbf{w})$ decrease monotonically? Or does it oscillate or even diverge?
- For GD:
 - Vary its learning rate $\eta > 0$. What happens if it is very small or if it is very big? Try to determine the value of η that gives the fastest convergence for your toy dataset by trial and error. Note: practical values of η are usually (quite) smaller than 1.
- For SGD:
 - Vary its learning rate $\eta > 0$ as for GD.
 - Vary the minibatch size $|\mathcal{B}|$ between 1 and N . How does this affect the speed of convergence (for fixed η)?
- For both GD and SGD we keep the learning rate η constant. How does this affect the behavior of SGD as we keep training? What should we do to improve that?
- Repeat all of the above in the following situations:
 - Use a dataset having 10 times as many points.
 - Use a dataset having 3 times larger noise.
 - Use a different initial \mathbf{w} .

Are the best values of η the same as before for GD? How about SGD? What other changes do you observe?

To visualize the results, create the following plots for each algorithm (GD and SGD):

- Plot the dataset (y_n vs \mathbf{x}_n) and the regression line $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$.
- Plot the error $E(\mathbf{w})$ over iterations, evaluated on the training set, and also on a validation set.
- Do a contour plot of the error $E(\mathbf{w})$ and the iterates $\mathbf{w}^0, \mathbf{w}^{(1)}, \dots$.

What you have to do: MNIST

- Select a small enough subset of MNIST as training inputs (using the whole dataset will be slow). Apply the ground-truth linear transformation to the data to generate the output labels $\mathbf{y} = \mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$. Then proceed as with the toy dataset to compute the optimal solution exactly by solving the normal equations, and approximately by running GD and SGD.
- Same questions as with the toy dataset. Note that the appropriate values for η , etc. may now be significantly different.

To visualize the results, create the following plots for each algorithm (GD and SGD):

- Plot the training and validation error $E(\mathbf{w})$ over iterations, as with the toy dataset.
- For each of the 4 results (true mapping, optimal mapping from the normal equations, and the mappings learned by GD and SGD), plot:
 - The parameters of the linear mapping using `imagesc(A)` and `imagesc(b)`. These are signed values, so use `colormap(parula(256))`.
 - If using as linear mapping the rotation/shift/scale/clip transformation, which produces as output a (possibly smaller) image \mathbf{y}_n , plot the following for a few sample images: the input image \mathbf{x}_n , and the outputs \mathbf{y}_n under the 4 mappings.

III (Stochastic) gradient descent for logistic regression

We consider the two-class case, given a sample $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$ with $\mathbf{x}_n \in \mathbb{R}^D$ and $y_n \in \{0, 1\}$. Again we assume the last element of each \mathbf{x}_n vector is equal to 1, so that w_D is the bias parameter in the sigmoid $\sigma(\mathbf{w}^T \mathbf{x}_n) = \frac{1}{1 + \exp(-(\mathbf{w}^T \mathbf{x}_n))}$. The objective functions $E(\mathbf{w})$ below do not admit a closed-form solution, so we need iterative algorithms. We apply GD and SGD, whose updates are obtained by computing the gradient $\nabla E(\mathbf{w})$ of E wrt \mathbf{w} . We can learn \mathbf{w} by maximum likelihood, or by regression.

By maximum likelihood We minimize the cross-entropy

$$E(\mathbf{w}) = \sum_{n=1}^N e(\mathbf{w}; \mathbf{x}_n, y_n) \quad \text{where} \quad \begin{cases} e(\mathbf{w}; \mathbf{x}_n, y_n) = -y_n \log \theta_n - (1 - y_n) \log (1 - \theta_n) \\ \theta_n = \sigma(\mathbf{w}^T \mathbf{x}_n) = \frac{1}{1 + \exp(-(\mathbf{w}^T \mathbf{x}_n))} \end{cases}$$

With GD and SGD we have the following updates:

$$\text{GD: } \Delta \mathbf{w} = \eta \sum_{n=1}^N (y_n - \theta_n) \mathbf{x}_n \quad \text{SGD: } \Delta \mathbf{w} = \eta \sum_{n \in \mathcal{B}} (y_n - \theta_n) \mathbf{x}_n.$$

By regression We minimize the least-squares error

$$E(\mathbf{w}) = \sum_{n=1}^N e(\mathbf{w}; \mathbf{x}_n, y_n) \quad \text{where} \quad \begin{cases} e(\mathbf{w}; \mathbf{x}_n, y_n) = \frac{1}{2} (y_n - \theta_n)^2 \\ \theta_n = \sigma(\mathbf{w}^T \mathbf{x}_n) = \frac{1}{1 + \exp(-(\mathbf{w}^T \mathbf{x}_n))} \end{cases}$$

With GD and SGD we have the following updates:

$$\text{GD: } \Delta \mathbf{w} = \eta \sum_{n=1}^N (y_n - \theta_n) \theta_n (1 - \theta_n) \mathbf{x}_n \quad \text{SGD: } \Delta \mathbf{w} = \eta \sum_{n \in \mathcal{B}} (y_n - \theta_n) \theta_n (1 - \theta_n) \mathbf{x}_n.$$

What you have to do Proceed as in the linear regression case. The only difference now is that the objective function $E(\mathbf{w})$ and hence its gradient and GD/SGD updates are different, and that the problem is classification rather than regression. Some additional questions to consider:

- Try datasets where the two classes are linearly separable, and where they are not linearly separable.
- In the linearly separable case, what happens to $\|\mathbf{w}\|$ as the number of iterations increases? Will the algorithm ever converge?
- Try using as initial weights $\mathbf{w} \in \mathbb{R}^D$ random numbers uniformly distributed in $[-u, u]$. Try a small value of u (say, 0.01) and a large one (say, 100). What happens, and why? Which initialization is better?

Once this works, you can implement logistic regression for the $K > 2$ classes case, if you feel adventurous.

Practical advice:

- Program thinking of n dimensions, not 2. It's more general and usually easier.
- Many operations in machine learning algorithms involve vectors and matrices. Try to program these using vector and matrix operations in Matlab ("vectorized code") rather than loops, because 1) the code will be shorter and more readable (closer to the pseudocode), 2) it will be faster (because Matlab is an interpreted language), and 3) you will save effort and avoid bugs. Example: a matrix-vector product $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$ instead of a double loop $y_i = \sum_{j=1}^n a_{ij} x_j$ for $i = 1, \dots, n$.
- Machine learning algorithms can have a high time or space complexity, so to get a result in a few seconds you may need to run them on small datasets. You can do this by selecting a random sample of a given dataset.
- Machine learning algorithms often are randomized. Likewise, toy datasets are usually generated randomly. To make sure you can generate the exact dataset multiple times and run an algorithm and get the same result every time, fix the seed of the pseudorandom number generator. In Matlab: `rng(1778);` where 1778 is the seed. You can also save a toy dataset for later use.
- Matlab tips:
 - To suppress extra line feeds: `format compact`.
 - To get more decimals: `format long`.
 - To compare two matrices or vectors (by finding the largest difference): `max(abs(A(:)-B(:)))`.
 - To avoid distorted plots: `daspect([1 1 1])`.
 - To plot grayscale images with values in $[0,1]$: `colormap(gray(256)); imagesc(I,[0 1]);`
To plot images with negative and positive values: `colormap(parula(256)); imagesc(I);`