

Stochastic Gradient Descent

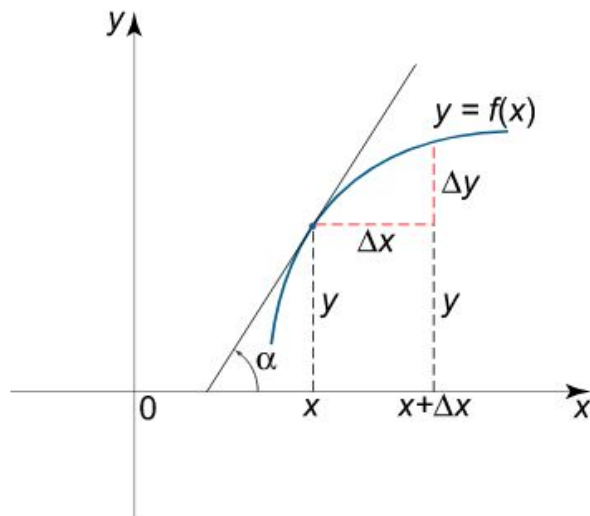
CS 189 Project T, Fall 2020
Team MJJER

Multivariable Calculus Concepts

Calculus Review

$$\frac{dy}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

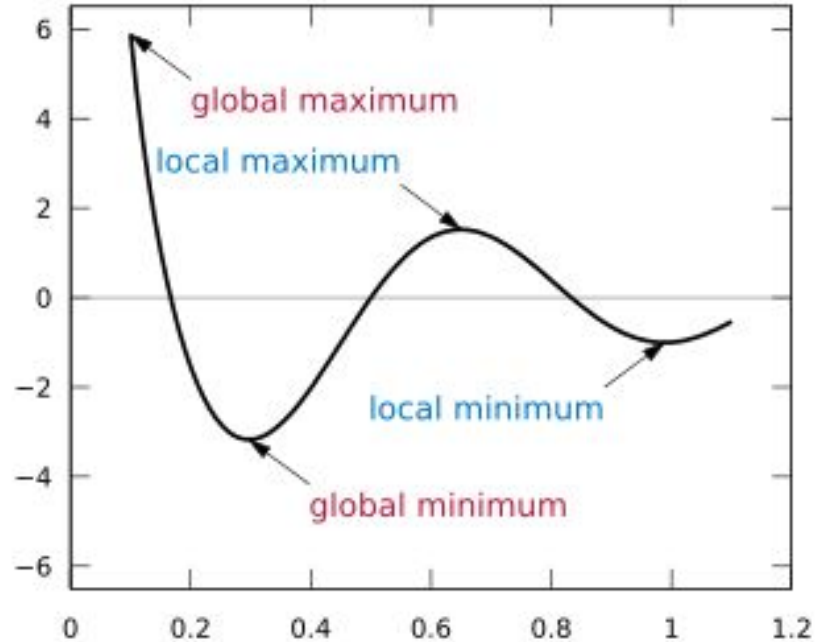
$$m = \lim_{x \rightarrow a} \frac{f(x) - f(a)}{x - a}$$



- Derivative represents the instantaneous rate of change
- Derivatives allow us to find the maximum and minimum value of a function

Critical Points

- By setting the derivative equal to zero we are able to find what are known as critical points
- Global maximum and minimum: maximum/minimum value across whole function
- Local maximum and minimum: maximum/minimum value across an interval or in an area “locally”
- Knowing exactly minimum or maximum requires the second derivative



Important Derivative Properties (Extra)

Linearity: Given $f(x)$ and $g(x)$ are differentiable functions and a and b are real numbers. The function $h(x) = a*f(x) + b*g(x)$ is also differentiable and the derivative is given by $h'(x) = a*f'(x) + b*g'(x)$.

Product Rule: Given the product of two functions, this rule will allow us to find the derivative of the product. Let $h(x) = f(x)g(x)$, then $h'(x) = f'(x)g(x) + g'(x)f(x)$.

Quotient Rule: The quotient rule gives us the derivative of one function divided by the other. Let $h(x) = f(x) / g(x)$, then $h'(x) = (f'(x)g(x) - g'(x)f(x)) / g(x)^2$

Chain Rule: The chain rule is used when we are trying to take the derivative of a composite function. Let $h(x) = f(g(x))$, then $h'(x) = f'(g(x))g'(x)$

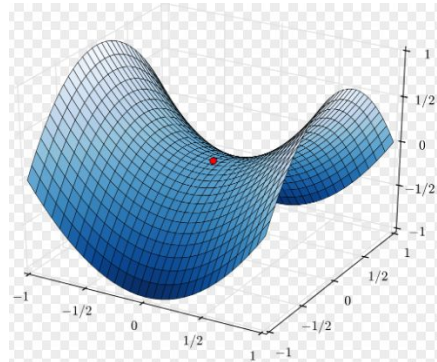
Gradients

-Take the derivative the same as we learned it for the 2 dimensional case by treating all the other variables as a constant and differentiating with respect to one variable.

$$\nabla f(x, y) = \begin{bmatrix} \frac{\partial}{\partial x} (x^2 - xy) \\ \frac{\partial}{\partial y} (x^2 - xy) \end{bmatrix} = \begin{bmatrix} 2x - y \\ -x \end{bmatrix}$$

Gradients

- Gradient vector is now represented as the direction and rate of fastest increase
- Called partial derivatives because taking the derivative with respect to only one variable at a time
- Introduces another critical point called the saddle point which is point on surface where all slopes in orthogonal directions are all zero but not a local extremum



Introduction to Gradient Descent

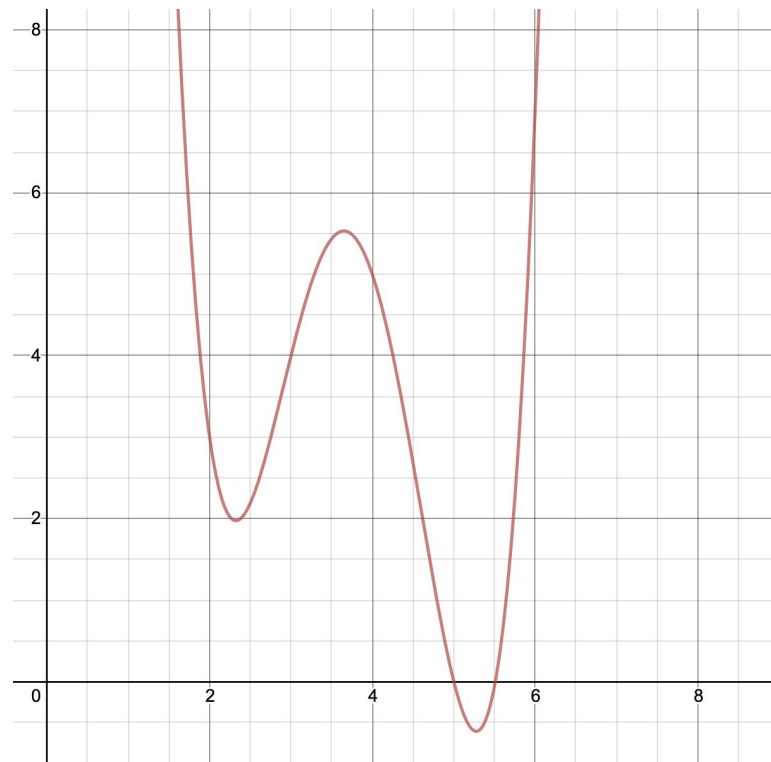
Introduction to 1D Gradient Descent

- GD is an optimization algorithm.
- Allows us to find minima of a function.
- Take this function for example:

$$y = x^4 - 15x^3 + 80x^2 - 179x + 145$$

- This could be a loss/cost function you need to minimize.
- Recall for regression we used loss:

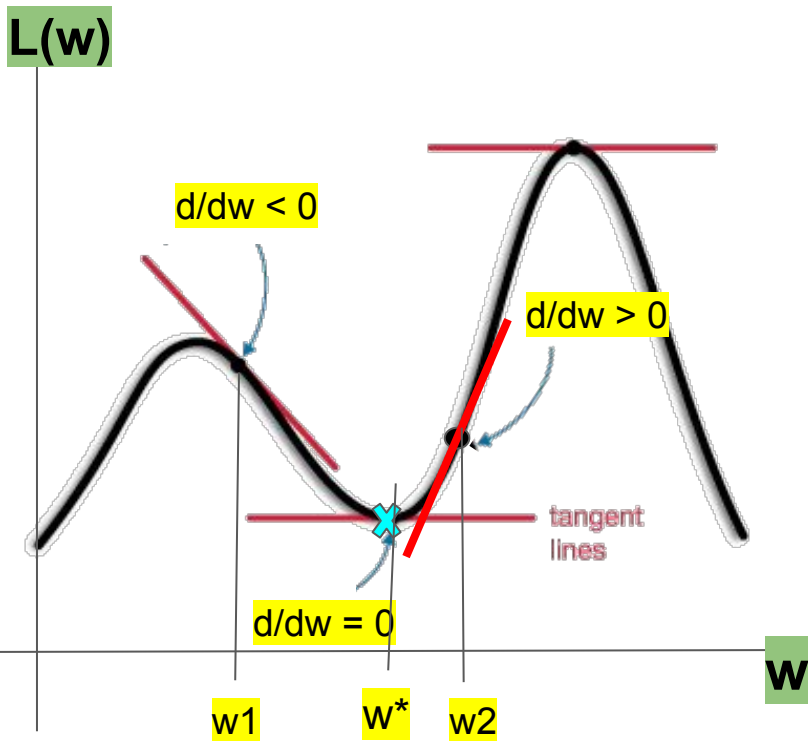
$$L(\mathbf{w}) = \sum_{i=1}^n (\mathbf{x}_i^\top \mathbf{w} - y_i)^2$$



Basic Intuition

We know how to find local and global minima for a 1D function.

We have to find and evaluate the points where the derivative is equal to 0.

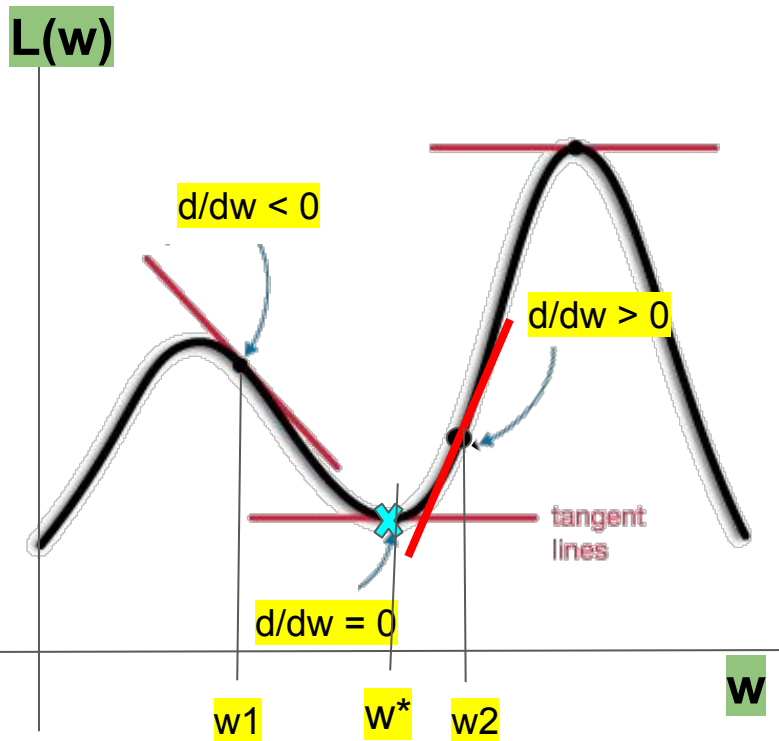


What does the derivative tell us?

1. If we are to the left of a minima, the derivative will be negative
2. If we are at a minima the derivative will be zero
3. If we are to the right of a minima the derivative will be positive

Introduction to GD Algorithm, 1D

The *sign* of the derivative tells us *which direction* we need to move to reach the minima and its *magnitude* tells us *how much* we need to move in that direction.

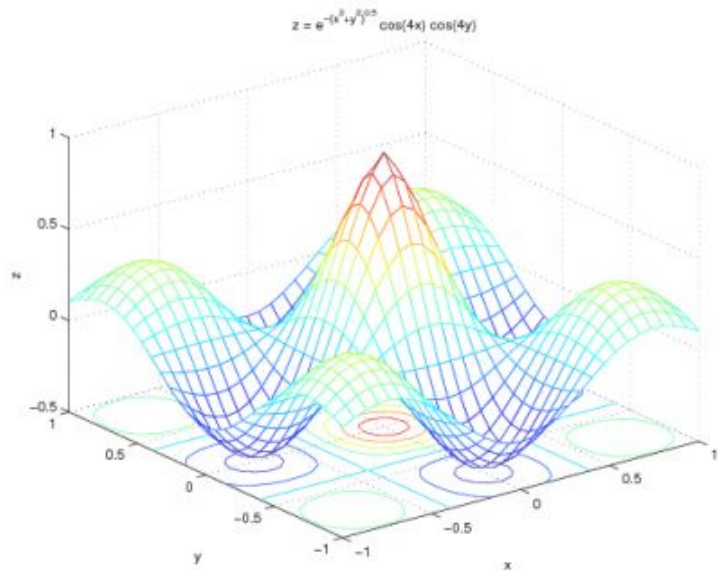


1. Start at a random point on the curve.
2. While d/dx is not 0:
3. Compute derivative at that point.
4. If the $d/dx > 0$ (we are to the right of the minima), take a step* to the left
5. If the $d/dx < 0$ (we are to the left of the minima), take a step* to the right

*the step should be proportional to the magnitude of the derivative. Notice that the closer you get to the minima point, the closer the magnitude of the derivative gets to zero.

Multidimensional Gradient Descent

- The same principles apply to the multidimensional loss functions we have seen in this class so far.
- Rather than computing the derivative, instead we need to compute the gradient of our loss function (C) with respect to the weights (w_1, \dots, w_n)



$$\begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

- We update our weights vector using the same update step as the 1D case.

$$\begin{bmatrix} w_1^+ \\ w_2^+ \\ \vdots \\ w_n^+ \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

The Algorithm

Initialize $\mathbf{w}^{(0)}$ to a random point

while $f(\mathbf{w}^{(t)})$ not converged **do** $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha_t \nabla f(\mathbf{w}^{(t)})$

```
1  def gradient_descent(compute_gradient, f, weights, alpha):
2
3      gradient = compute_gradient(f, weights)
4
5      while gradient != 0:
6          gradient = compute_gradient(f, minima)
7          weights = weights - alpha * gradient
8
9      return weights
```

Learning Rate, “Step Size”, alpha

What is the learning rate?

- The learning rate allows us to scale the gradient to control our step size in the update step

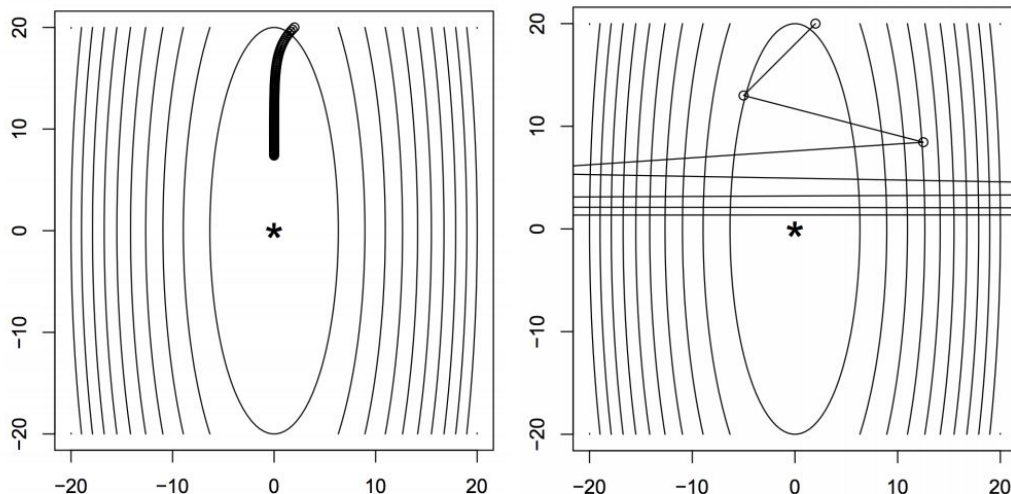
Why is it necessary?

- A scaling that is **too high** may cause divergence from the optimal solution.
- A scaling that is **too low** may cause the algorithm to converge too slowly.

How do we set it?

- Setting the learning rate is dependent on attributes of the loss/cost function
- Additionally, it is possible to use an adaptive step size. These ideas will be explored in the coding notebook.

Step Size, Divergence, Runtime



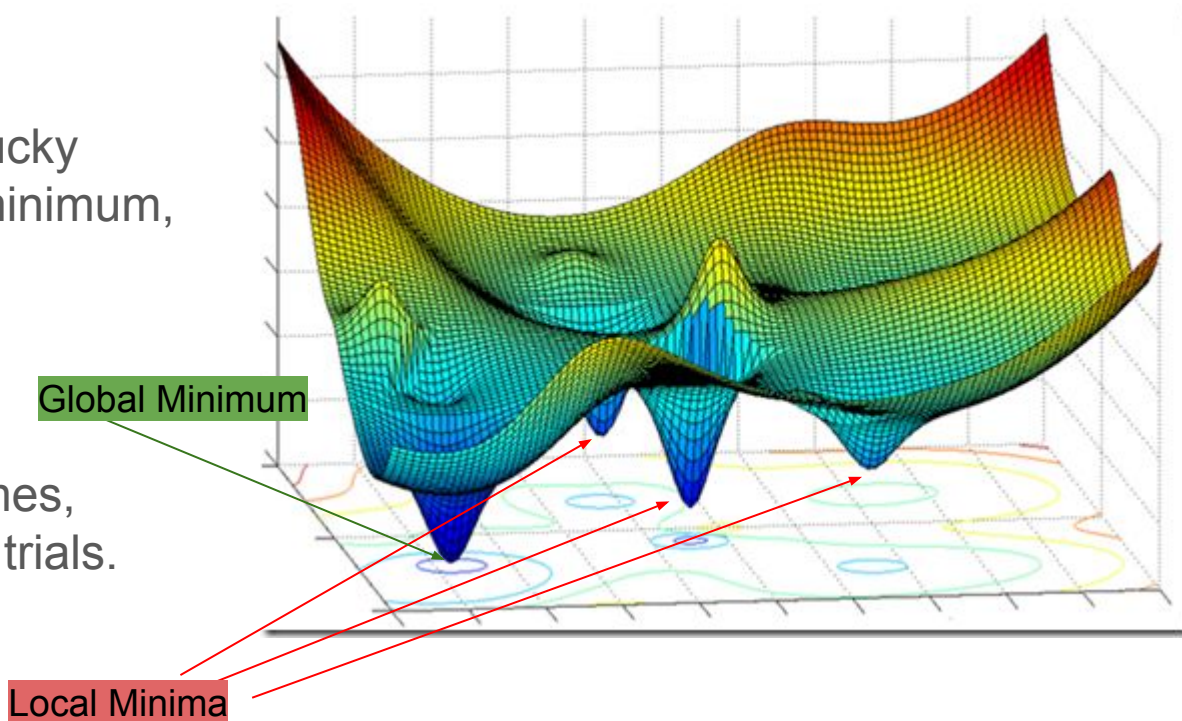
Setting the learning rate too small will slow down the algorithm too much, shown visually on the left. The algorithm may never converge to the optimal point.

Setting the learning rate too large will result in the step size being too large, and the algorithm might just “step over” the optimal point and miss it entirely. This is shown to the right.

Limitations of Gradient Descent I: Local Minima

The GD Algorithm finds local minima, we could get lucky and end up with the global minimum, but this is not guaranteed.

A potential way to improve is to reinitialize the weights
And re-do the algorithm n times,
And take the best of those n trials.



Limitations of Gradient Descent II: Efficiency

- As we have seen in this class, we are mostly dealing with loss functions of this nature:

$$f(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n f_i(\mathbf{w})$$

- We are individually computing losses over n training points and taking the average. Thus, the gradient of our loss looks like a sum over n points.

$$\nabla f(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{w})$$

- This means we are computing n gradients everytime we compute the gradient in the while loop of our gd algorithm. This is computationally expensive!
- This is the standard form of the gd algorithm called *batch gradient descent*.
- Can we avoid computing n gradients every single time?

Potential Improvement: “Mini-Batch”

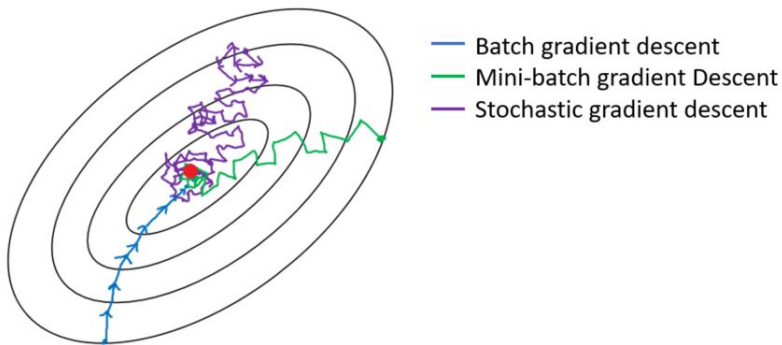
$$\nabla f(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{w}) \longrightarrow G(\mathbf{w}) = \frac{1}{k} \sum_{i=1}^k \nabla f_i(\mathbf{w})$$

- Instead of computing all n gradients (computationally inefficient). Just compute k of them where $k < n$.
- “On average” $G(\mathbf{x})$ will be a “pretty decent” estimate of $\nabla f(\mathbf{w})$ and we saved ourselves from computing $n-k$ gradients every time.
- Another improvement is that the gradients of mini-batch are more “noisy”, so we are potentially decreasing the probability of quickly getting stuck in a saddle point or local minima.

Stochastic Gradient Descent

Definition

- SGD approximates the gradient using a single randomly chosen data point for each epoch
- Equivalent to mini-batch gradient descent where $k = 1$



Advantages of SGD

- SGD is useful for higher computation speed since we only examine one training point at a time
- Frequent updates allow immediate insight into the model, especially on large datasets
- More noise = less susceptible to local minima
- Easier to fit into memory
- Preferred for most large datasets where computing n gradients for each update is too computationally expensive

Disadvantages of SGD

- By definition, SGD has lower accuracy due to randomness of training point selection at the minimum
- Oscillates around the minimum and doesn't truly converge, leading to greater variance over the measured loss from run to run
- Less computationally efficient than mini-batch gradient descent
- Takes longer to fully train the model

Bounds on Convergence

- SGD works well with convergence on earlier steps, but not with later steps
- Large step sizes cause the algorithm to diverge
- Theoretically, we may use diminishing step sizes for SGD so that earlier steps cause large changes in the parameters and later steps can converge upon the minimum
- However, constant step sizes that are sufficiently small will allow for exponential convergence

Implementing SGD in sklearn, Tutorial

SGD in Sklearn

- Sklearn library contains a lot of efficient tools for machine learning and statistical modeling including classification, regression, clustering, and dimensionality reduction
- Interfaces Python
- Common sklearn machine learning problem skills
 1. Loading data set
 2. Learning the model
 3. Predicting for the new data

SGDRegressor

- Linear model fitted by minimizing a regularize empirical loss with SGD
- Important parameters:
 1. loss: loss function to be used
 2. penalty: regularization term to be used
 3. alpha: constant that will multiply the regularization term
 4. learning_rate: the learning rate
 - a. 'constant' means $\eta = \eta_0$
 - b. 'optimal' means $\eta = 1.0 / (\alpha * (t + t_0))$

$$w \leftarrow w - \eta \left[\alpha \frac{\partial R(w)}{\partial w} + \frac{\partial L(w^T x_i + b, y_i)}{\partial w} \right]$$

Additional Sklearn Functions

`sklearn.preprocessing.StandardScaler`: This will transform our data distribution to have a mean value of 0 and a standard deviation of 1.

`sklearn.pipeline.make_pipeline`: This will construct a pipeline for a given estimator.

There are many more variations in sklearn that you can learn more about in the [sklearn documentation page](#). The general ideas behind these functions, however, are the same.

Loss Functions

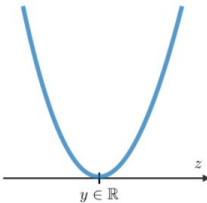
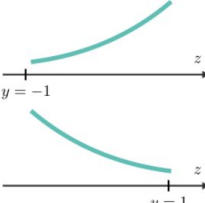
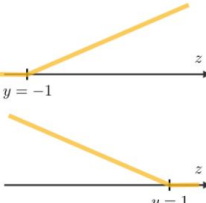
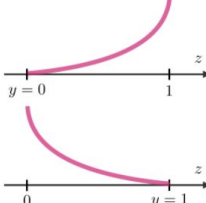
- A loss function is a function that measures the error between the true output and an estimate of the true output.
- Mathematically, this is denoted as,

A loss function is a function $L : (z, y) \in \mathbb{R} \times Y \mapsto L(z, y) \in \mathbb{R}$

- SGD computes the gradient of this loss at each iteration of the algorithm where we want to take on the direction of minimal loss.
- We want to explore how depending on the model, we observe different performance for SGD.

Loss functions for different models.

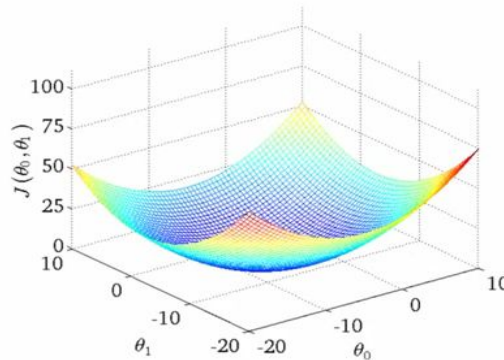
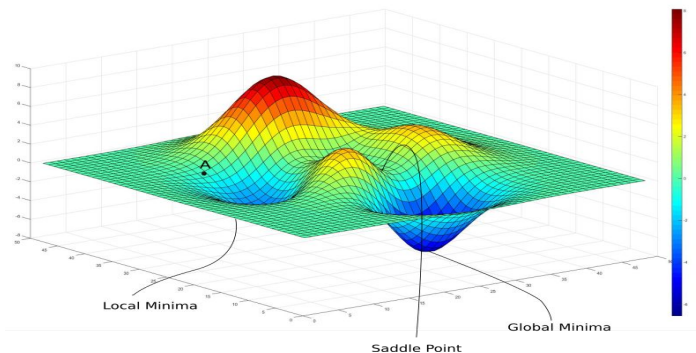
- The choice of a loss function depends on the type of model we want to represent our problem with e.g regression or classification.
- Some examples of Loss functions are pictured below with corresponding models:

Least squared error	Logistic loss	Hinge loss	Cross-entropy
$\frac{1}{2}(y - z)^2$	$\log(1 + \exp(-yz))$	$\max(0, 1 - yz)$	$-\left[y \log(z) + (1 - y) \log(1 - z)\right]$
			
Linear regression	Logistic regression	SVM	Neural Network

- SGD has different patterns of convergence for each loss function and also varies in the estimation errors.

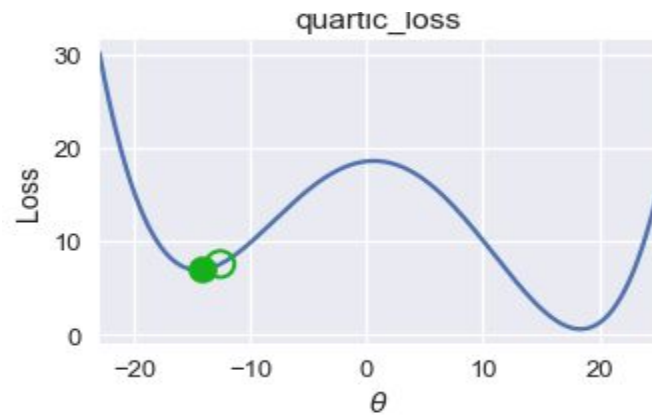
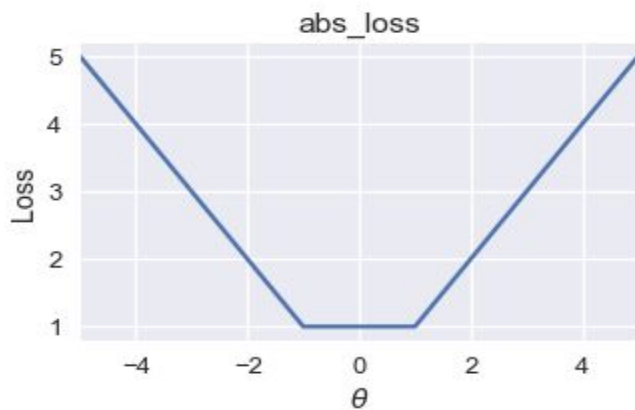
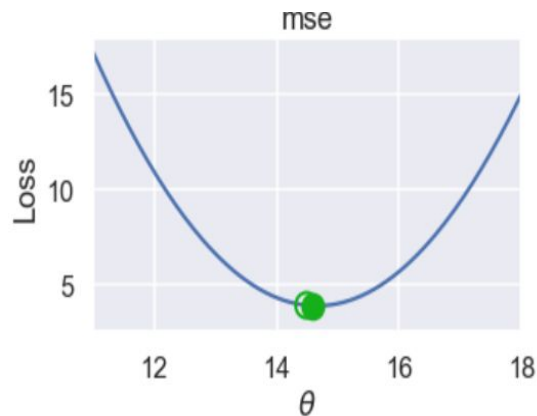
Saddle Points, Local Minima vs Global Minima

- Loss functions are susceptible to the convergence of local minima that may not be the optimal solution we want.
- Saddle points are points of zero gradient which causes SGD to get stuck.
- This is due to the convexity of the loss function where there exist many local minima.
- To help SGD achieve convergence to global minima we would want our loss function to be convex.



Examples of loss functions with local minima, global minima and saddle points.

- The quartic loss function misses the global minima as it converges to a local minima under SGD we observe that this loss function is not convex.
- MSE given an appropriate learning rate will converge to global minimum as this loss function is convex.
- Abs Loss has a saddle point but in this case it is also the global minima in the interval $[-1, 1]$.

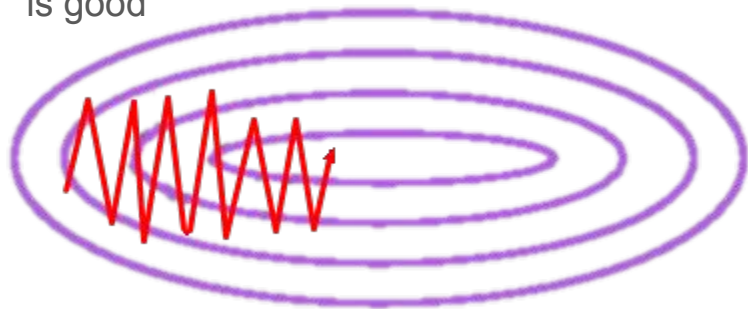


Variants and Optimizations on SGD

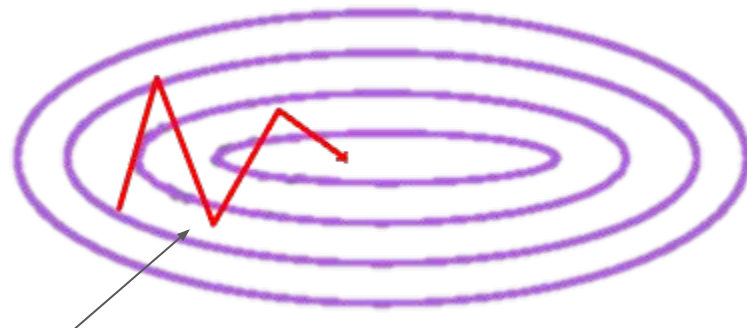
Momentum and Acceleration

Motivation

- Gradient Descent - only cares about gradient at a given point
 - Very easy to get stuck in local minima, as gradient approaches 0.
- How do we not get stuck in smaller “valleys” of the loss function?
- Momentum: incorporate the “speed” we had from the previous gradient update step.
 - Our update depends on the gradient and our “velocity” from the previous update.
 - Velocity represents how much our weight estimate changes with each time step.
- We also should not overshoot the optimum with our momentum term - slowing down near optimum is good



Regular SGD



Maintains “forward velocity”

SGD with Momentum

Momentum-based Gradient Descent

Velocity

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - v_t$$

Standard gradient
Descent update

A high velocity
could cause us to
overshoot the
optimum

Update weight using velocity
(changes in position = velocity)

We take the previous velocity, and weigh it with a discount factor gamma (usually 0.9). This acts as an exponential moving average - the previous step is weighted by gamma, two steps ago by gamma squared, etc. So at a given point, we have a dual-effect: the previous velocity and the current gradient.

Nesterov's Accelerated Gradient Descent

- A clever modification on Momentum-based Descent that fixes the issue of overshooting optimum with very high velocities
 - Allows us to slow down if we “look ahead” and see a local minimum.

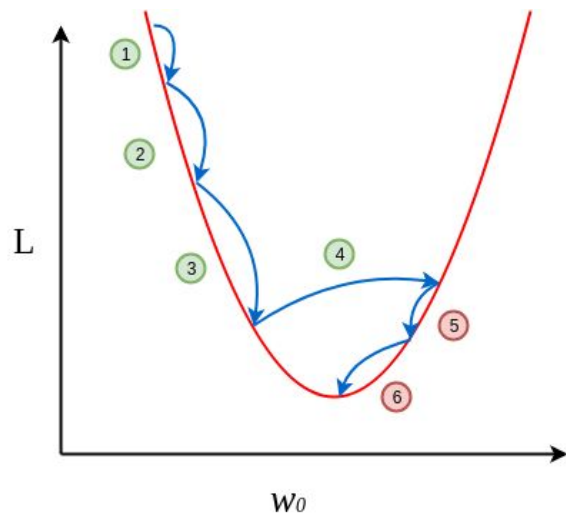
$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})$$



The gradient we IF updated the weight using momentum-based descent.

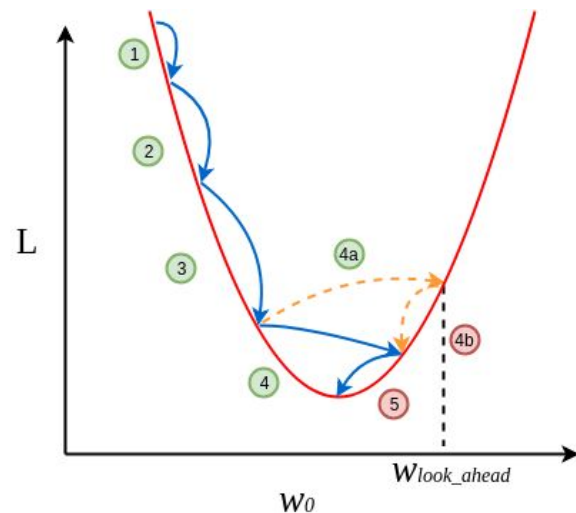
- If the “look-ahead” gradient is about in line with momentum update, we get a “boost.”
- If the “look-ahead” gradient points opposite to the momentum update (usually occurs near an optimum), our rate of descent slows.

A Quick Illustration



(a) Momentum-Based Gradient Descent

$$\text{Green Circle} \Rightarrow \frac{\partial L}{\partial w_0} = \frac{\text{Negative}(-)}{\text{Positive}(+)}$$



(b) Nesterov Accelerated Gradient Descent

$$\text{Red Circle} \Rightarrow \frac{\partial L}{\partial w_0} = \frac{\text{Negative}(-)}{\text{Negative}(-)}$$

Adaptive Learning Rate Methods

Motivation

- Our optimization is over several directions - we may need more delicate updates in certain directions, and more aggressive updates in others

$$x_1 \leftarrow x_1 - \eta \frac{\partial f}{\partial x_1}, \quad x_2 \leftarrow x_2 - \eta \frac{\partial f}{\partial x_2}$$

- If the x_1 component has a smaller gradient, then we would like a larger learning rate in the x_1 direction.
- If the x_2 component has larger gradients, then we want to be more careful with our steps - a smaller learning rate would be more ideal.

RMSProp

- Idea: Maintain a vector of learning rates - one for each component - that is an exponentially moving average of the norm past gradients.

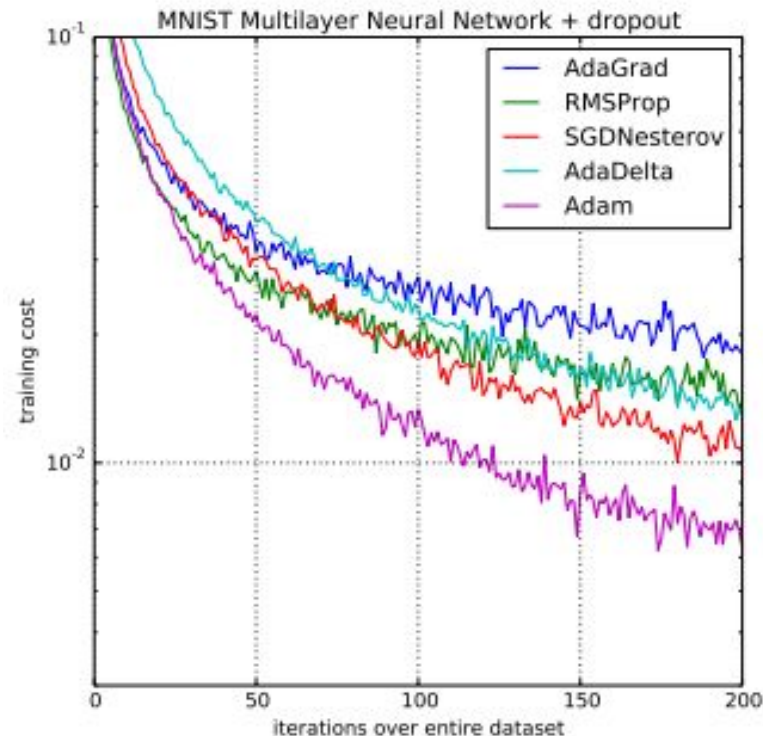
The diagram illustrates the RMSProp algorithm with two equations and several annotations:

- Equation 1:**
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta_t}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t$$
 - An arrow points from the text "Gradient at time t" to the term \mathbf{g}_t .
 - An arrow points from the text "Element-wise Square-Root" to the denominator $\sqrt{\mathbf{s}_t + \epsilon}$.
 - An arrow points from the text "Small constant to avoid Numerical Instability" to the term ϵ .
- Equation 2:**
$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t \odot \mathbf{g}_t$$
 - An arrow points from the text "Moving average of the Norm^2 of gradients" to the term \mathbf{s}_t .

- By dividing by a discounted average of the norm past gradients, directions with smaller gradients in past updates will get a boost (dividing by a small number), and directions with larger gradients will slow their descent.

ADAM - Adaptive Moment Estimation

- State-of-the-Art Optimizer that combines Momentum and an Adaptive Learning Rate (RMSProp)
- Maintains an exponentially moving average of both velocity and the norm squared of past gradient.
- See Coding Exercise on how to implement ADAM.



Works Cited

EECS 189 Note 12: <https://www.eecs189.org/static/notes/n12.pdf>

Data 100 Slides:

https://docs.google.com/presentation/d/1gi7Ar5O7T0qE_abZeZvX-VURC-i3sWwwwK6FtsulwkQc/edit#slide=id.g8c5b259fcb_0_1306

Sklearn SGDRegressor documentation:

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html

Momentum, Adagrad, and Adam Optimization Overview:

<https://runder.io/optimizing-gradient-descent/>