# Stochastic Gradient Descent

Justin Park, Meera Mehta, Emily Teng, Joshua Gonzalez, Rohith Sajith

Project T Final, December 1 2020

## 1 Calculus Review - Derivatives

### 1.1 Notation

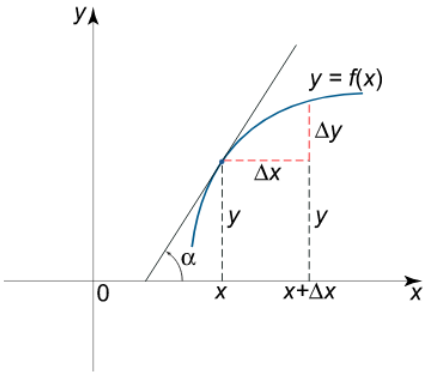
The derivative of the function $f(x)$ with respect to $x$ is commonly depicted by $f'(x)$ or $\frac{\mathrm{d}f}{\mathrm{d}x}$.

### 1.2 Definition

The formal definition of a derivative of $f(x)$ with respect to x is the function $f'(x)$ and is defined as,

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In the definition above, the numerator represents the change in y value and denominator represents the change in x value. By taking the limit of the change to 0, we get the instantaneous rate of change of the function. Let's now see a visual example on different graphs. The graph below, the blue line represents the function and the black line, which is tangent to the function, represents the derivative at that specific point.



### 1.3 How to take the derivative?

For the purposes of this topic, we will show a mechanical review of derivative of functions.

| | | |
|---|---|---|
| Constant | c | 0 |
| Line | $x$ | 1 |
| Square | $x^2$ | $2x$ |
| Square Root | $\sqrt{x}$ | $\frac{1}{2}x^{-}(\frac{1}{2})$ |
| Exponential | $e^x$ | $e^x$ |
| Logarithms | $\log x$ | $\frac{1}{x}$ |

### 1.4 Properties

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 $h'(x) = a * f'(x) + b * g'(x)$. This is known as the linear combination rule.

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

$\frac{d}{dx}\left(f\left(x\right)g\left(x\right)\right) = f\left(x\right)\frac{d}{dx}g\left(x\right) + \frac{d}{dx}f\left(x\right)g\left(x\right)$

Quotient Rule: The quotient rule gives us the derivative of one function divided by the other. Let $h(x) = \frac{f(x)}{g(x)}$, then

$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{\frac{d}{dx}f(x)g(x) - f(x)\frac{d}{dx}g(x)}{g^2(x)}$

Chain Rule: The chain rule is used when we are trying to take the derivative of composite function. In other words, taking derivatives of a function of a function.

$\frac{d}{dx}\left[f\left(u\right)\right] = \frac{d}{du}\left[f\left(u\right)\right]\frac{du}{dx}$

## 2 Why are derivatives important?

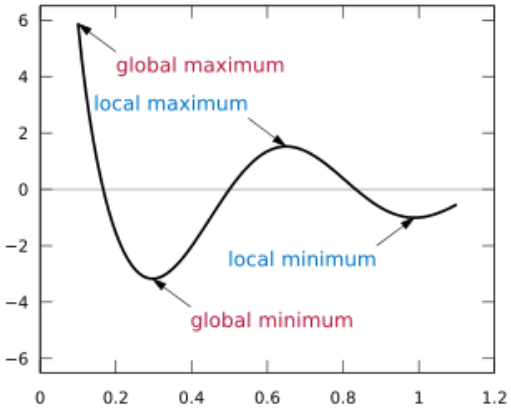
Derivatives ultimately tell us the slope of a particular point in a graph. Derivatives allows us to find the maximum and minimum point in a function. In the following section, we will explore different types of points in a function and see how we can use a more general concept of gradients to find a minimum or maximum point of a function. In particular, we will focus on minimizing loss functions which will be discussed later in the notes.

### 2.1 Minimum and Maximum Points

Global Minima and Global Maxima: Global minima is the minimum value across the whole function and the global maxima is the maximum value across the whole function.

Local Minima and Local Maxima: The local minima and local maxima represent the places where the function reaches minimum and maximum value in a

certain interval. It may not be maximum and minimum value across the whole function but locally it is the minimum and maximum values.




The image above highlights the different types of points nicely.

# 3 Gradients

Now that we talked about derivatives in 2 dimensions, we will now discuss derivatives on higher dimensions. The gradient is the dual of the derivatives on higher dimensions. Now intead of a tangent to the function, it represents a tangent vector. The gradient vector is now represented as the direction and rate of fastest increase.

## 3.1 Notation

The gradient of a function is a vector denoted as $\nabla$.

## 3.2 How to take the gradient?

To take the gradient of a multivariable function, take the derivative the same as we learned it for the 2 dimesnional case by treating all the other variables as a constant and differentiating with respect to one variable. Then repeat this for all the variables and you should have a vector of partial derivatives. This partial derivative is denoted by

$$\textit{First order partial derivative} = \frac{\partial f}{\partial x}$$

$$\textit{Second order partial derivative} = \frac{\partial^2 f}{\partial x^2}$$

$$\textit{Third order partial derivative} = \frac{\partial^3 f}{\partial x^3}$$

$$\vdots$$

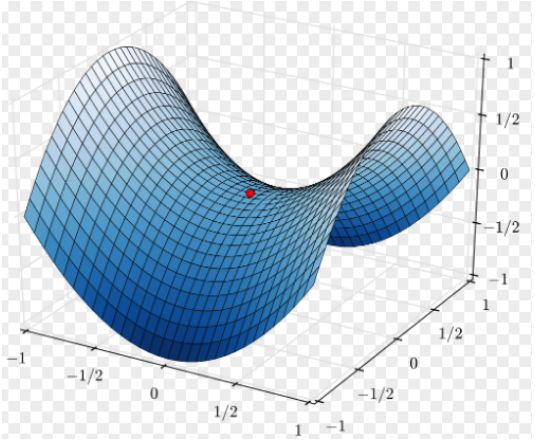
$$Kth\ order\ partial\ derivative = \frac{\partial^k f}{\partial x^k}$$

and the gradient vector with respect to two variables is denoted by

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial l}{\partial x_2} \end{pmatrix}$$

### 3.3 Saddle Points

Now in multivariable calculus, there is a new point called the saddle point. This is the point on the surface of the graph of a function where the slopes in orthogonal directions are all zero but not a local extremum.




Why is this important? As we begin trying to optimize our loss functions to higher dimensions, we are trying to find the places where the derivative is zero which are also known as critical points. These critical points are the saddle points.

## 4 Introduction to Gradient Descent

### 4.1 1D Gradient Descent

Gradient descent is an optimization algorithm that allows us to find the minimum of a function by stepping towards a critical point using the basic calculus principle of derivatives/gradients that we have reviewed above. In machine learning, as we have seen in the past few weeks, our goal has often been to find a scalar or a vector of "weights" that minimize a cost/loss function.

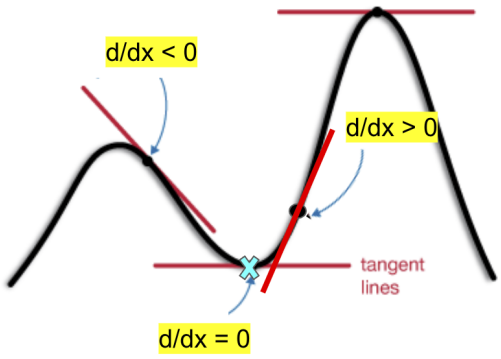
For example, recall from EECS 16A, ordinary least squares, where our goal is to minimize the loss between $\hat{y} = X * w$ and $y_{test}$ . Which can be the mean squared error function: $L(w) = \frac{1}{N}\sum_{i=1}^{n}(x_i^T w - y_i)^2$ or the L2 loss function:

$L(w) = \frac{1}{2}\sum_{i=1}^{n}(x_i^T w - y_i)^2$.

Let's consider the scalar case, where we have a scalar weight and we are trying to find the value of w which minimizes our loss L(w). We know how to do this using basic calculus for a 1D function but doing this may be computationally inefficient when working with very complicated multidimensional loss functions. That's where gradient descent comes in.

Gradient descent can be thought of as choosing a random point on a hill and putting a ball there. The ball will move in the steepest downwards direction until it settles into a local minimum.

In the scalar case, to find which direction to move in, we can look to the derivative. If you choose a point to the right of a local minimum, then the function will be sloping upwards there, and thus, the derivative will be positive and to the left of a local minimum, the derivative will be negative. At the local minimum, the derivative is zero. See figure below.




Thus the derivative at any point w tells you the direction you need to step in and how much in that direction you need to step. Namely, to the right of a local minimum you need to decrease your w value to get closer to the minimum (subtract the positive derivative from your w value). When to the left of a local minimum you need to increase your w value to get closer to the minimum (subtract the negative derivative from your w value). This simple update step is the key to gradient descent: w(t+1) = w(t) - a*dL/dw (a is the learning rate, we will discuss this in section 4.3). Also, notice that when getting closer to the local minima, the magnitude of your derivative decreases, this means your step size will also decrease, as you want to avoid overshooting and passing the local minimum.

Initialization is easy, just choose any point, w, to start from. Now, with every iteration of the algorithm, one must recompute the gradient, and update

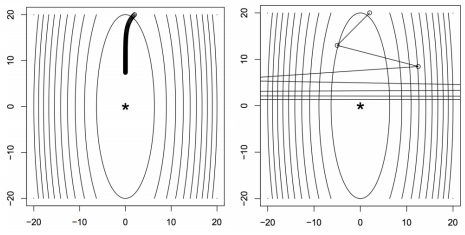
the weights. When does the algorithm converge? The algorithm converges when the weights are no longer updated after an update step. Notice that this occurs when the derivative at w is zero, which means you have reached a local min.

## 4.2 Multidimensional Gradient Descent

This is very easy to extend to the multidimensional case. Instead of a scalar w, we have an n-dimensional vector w. This means instead of a 2D graph we have an n-dimensional graph. We can initialize the weights vector randomly, as before. Instead of computing the derivative of the loss function, we now compute the gradient. Recall that the gradient is just a vector where each entry contains the partial derivative of the loss function with respect to one of the n weights.

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

## 4.3 Learning Rate, and the Algorithm



[1]

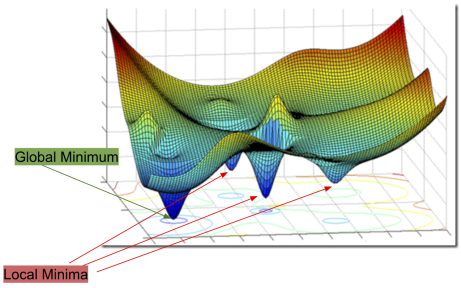
Notice in the update step above there is a constant that scales the gradient before it is applied to the weights vector. This constant is the learning rate/step size. It is useful because if we set it intelligently based on attributes of our loss function we can leverage this extra parameter to achieve faster and better results. Some issues that arise with the learning rate are setting it to be too large, which can result in the algorithm missing the local minima altogether. Additionally, if it is too small the algorithm may take too long to run and never converge. The basic idea of 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]

## 4.4 Limitations of Gradient Descent

The gradient descent algorithm only finds local minima, not the global minimum. Thus it is useful to run the algorithm many times at different initialization points and take the best of these to potentially get a better result. Additionally, we may be able to use domain knowledge to choose our initialization point more efficiently. Another limitation arises when our loss function involves summing over n training points.




This means our gradient will also involve a summation over each of the n training points. This is potentially very computationally expensive since we are computing the n gradients and taking the average of them at every iteration of the algorithm. If we compute the gradient above, this called batch gradient descent.

## 4.5 Brief Introduction Mini-Batch Gradient Descent

There is a clear way to improve the computational efficiency of batch gradient descent. Namely, decreasing the batch size. Regularly, using all points we would have:

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

Instead, in mini-batch gradient descent. For a $k < n$:

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

Instead of computing n gradients, we just choose k of the indices. On average, G(x) will be a pretty decent estimate of the gradient. Another improvement is that the gradients of mini-batch are more "noisy", so we are potentially decreasing the probability of getting stuck in a saddle point or local minima. We will go into more detail in the next section.

# 5 Stochastic Gradient Descent

## 5.1 What is SGD?

[4]
Stochastic gradient descent is a type of gradient descent that uses a single randomly chosen data point to approximate the gradient for each parameter update. Instead of having to compute $n$ gradients for each update and averaging over all the gradients, we only have to compute a single gradient every time. This is equivalent to mini-batch gradient descent in the case where $k = 1$.

Stochastic gradient descent is useful for large data sets where taking the gradient of $n$ data points for each update would be too computationally expensive. We can compare the equations for batch gradient descent, mini-batch gradient descent, and stochastic gradient descent, where $\omega_t$ is the estimate of our model parameter at time $t$, $\alpha$ is the learning rate, and $n$ is the number of training points in the data set.

Batch gradient descent:

$$\omega_{t+1} \leftarrow \omega_t - \alpha_t * \frac{1}{n} \sum_{i=1}^{n} \nabla(f_i(x))$$

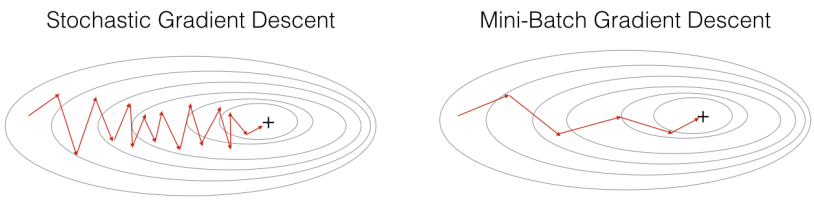
Mini-batch gradient descent for $k < n$:

$$\omega_{t+1} \leftarrow \omega_t - \alpha_t * \frac{1}{k} \sum_{i=1}^{k} \nabla(f_i(x))$$

Stochastic gradient descent for a randomly chosen point $i$:

$$\omega_{t+1} \leftarrow \omega_t - \alpha_t \nabla(f_i(x))$$

For stochastic gradient descent, it is important to choose the data point randomly for each update. Often, the data will be shuffled and the algorithm will perform a full pass over the shuffled data, updating the parameters $n$ times; then, the data will be reshuffled for the next pass.




[4]

## 5.2 Advantages and Disadvantages of SGD

The motivation behind SGD is to reduce the computational cost of each update. Thus, the primary advantages of SGD are higher computation speed and reduced memory usage. Compared to batch gradient descent, SGD is more useful for large data sets where the number of points is too high to efficiently calculate the gradient of all points for every update. SGD can perform frequent updates that only examine a single training point at a time.

SGD is also called "online" gradient descent because it utilizes the data in a certain (randomized) order to make predictions about a model for future data that has yet to be observed. Thus, its frequent updates allow immediate insight into the model as the data is observed. This is again useful for large data sets because the data set does not have to be fully "processed" for the model to be trained.

Like mini-batch gradient descent, the increased noise in SGD makes it less susceptible to local minima; it is able to escape these because of a higher level of fluctuation.

The disadvantages of SGD are a result of both its lower accuracy due to the selection of a single point for each update and the amount of updates required to cover the entire data set. There will be greater variance in the results between runs due to greater noise. Also, since SGD only examines one point per update, it would require $n$ updates to complete a single pass over the data. Thus, SGD may end up being less computationally efficient than mini-batch gradient descent, which trades off some of the speed of SGD for the ability to perform a smaller amount of updates with better accuracy.

## 5.3 Learning Rate

One important problem in SGD and machine learning overall is the issue of choosing a learning rate $\alpha$, or the step size. Setting the learning rate too high can cause the algorithm to diverge, while setting it too low can cause the algorithm to converge too slowly.

SGD causes additional complications with convergence upon the exact mini-

mum; large fluctuations due to the randomness in the algorithm mean that SGD tends to converge to a "noise ball" as it performs updates close to the minimum. Thus, SGD performs better than batch gradient descent on earlier steps when it is far from the minimum, and worse than batch gradient descent on later steps as it approaches the minimum.

In practice, a high level of accuracy near the minimum is sometimes not necessary, so a constant learning rate is used. One simple extension of SGD makes the learning rate a decreasing function of time, reducing the step size on each update so that earlier steps cause large changes and later steps may converge more precisely upon the minimum. As we will see in later sections, there are various ways to optimize SGD and improve the rate of convergence.

# 6 Introduction to scikit-learn(sklearn)

[5]

## 6.1 What is sklearn?

The sklearn library contains a lot of efficient tools for machine learning and statistical modeling including classification, regression, clustering, and dimensionality reduction. It is interface by Python. These are some of the most common functions used in sklearn. Feel free to look into the sklearn documentation page to learn more.

## 6.2 The machine learning problem

We will typically have a learning problem where we will use a data set to predict properties of the unknown data. There are many different types of learning problems but for now we will just go over the general ingredients needed to solve this problem.

1) Loading an example data set: sklearn comes with some standard data sets which we could utilize like so.

```
$ python
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> digits = datasets.load_digits()
```

2) Learning and predicting: Now given the data, we can learn a model which we can then use to predict something. For example, given a digits data set, our goal could be to predict which digit an image represents.

In sklearn, an estimator implements the fit(X,y) and predict(T) methods. Given a classifier "clf" we can fit and predict like so.

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, gamma=0.001)
```

```
>>> clf.predict(digits.data[-1:])
array([8])
```

# 7 How to implement SGD in sklearn?

Let's now dive more specifically on how to utilize sklearn for SGD.

## 7.1 Mathematical Formulation

SGD is an optimization method for unconstrained optimization problems by approximating the gradient by considering a single training example at a time. We will introduce $sklearn.linear_model.SGDRegressor$ function which is a linear model fitted by minimizing a regularized empirical loss with stochastic gradient. It will first estimate the gradient of the descent of the loss function at each of the samples and update the model along the way depending on the learning rate.

## 7.2 Important Parameters

*loss*: The loss function to be used.

*penalty*: The regularization term to be used.

*alpha*: Constant that will multiply the regularization term. This is also used to compute the learning rate.

$learning_rate$: The learning rate. 'constant' means eta=eta0. 'optimal' means eta = 1.0 / (alpha * (t +t0)) eta0: The initial learning rate for the 'constant'.

## 7.3 Additional Functions

$sklearn.preprocessing.StandardScaler$: This will transform our data distribution to have a mean value of 0 and a standard deviation of 1. This will be done feature wise independently. The reason we need to normalize our features is because we want different features to take on similar ranges of values so that the gradient will converge more quickly.

$sklearn.pipeline.make_pipeline$: This will construct a pipeline for a give estimator. This is not necessary but it allows us to concisely process and fit our data in one function without having to repeat them individually.

### 7.4 Example Implementation

```
>>> import numpy as np
>>> from sklearn.linear_model import SGDRegressor
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> # Always scale the input. The most convenient way is to use a pipeline.
>>> reg = make_pipeline(StandardScaler(),
...                     SGDRegressor(max_iter=1000, tol=1e-3))
>>> reg.fit(X, y)
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('sgdregressor', SGDRegressor())])
```

1) We import the necessary sklearn functions.
2) We create random X samples and random Y samples.
3) Create a pipeline that will first scale the features then run SGDRegressor.
4) Use the pipeline we created to fit the model.

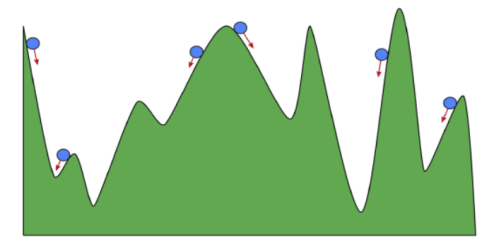
Now at this point, we have a Pipeline object that will allow us to make predictions for our linear model. We simply use the *predict*() method with features as the input.

### 7.5 Conclusion

There are many more variations in sklearn that you can learn more about in the sklearn documentation page. This is an example of using SGDRegressor for linear regression but we could also perform classification using SGDClassifier. The general ideas behind these functions, however, are the same.

## 8 Effects of Different Loss Functions on SGD

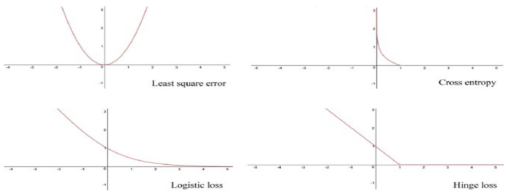
We know that depending on the loss function used we obtain different behavior for both tasks of regression and classification using SGD. We first note the issue of convexity of a loss function. The gradient descent algorithm relies on this property to obtain convergence although it is not a necessary condition. If this property does not hold then SGD may converge to local minima or a saddle point. We observe this with the following diagram:



Luckily, several of the loss functions used are convex so SGD is guaranteed to have convergence to its global minima, we see some examples below with

some of their graphs.

- All of these are convex upper bounds on 0-1 loss.
- Hinge loss: $L(y, \hat{y}) = \max\{0, 1 - y\hat{y}\}$
- Exponential loss: $L(y, \hat{y}) = \exp(-y\hat{y})$
- Logistic loss: $L(y, \hat{y}) = \log_2(1 + exp(-y\hat{y}))$

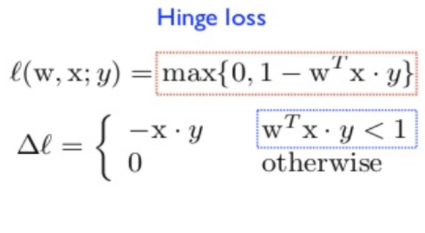



The SGD requires us to compute the gradient of an appropriate loss function so then we explore properties of smoothness of such loss functions. From visual observation, we notice that the graphs of various functions are sufficiently smooth and we can easily obtain the derivative but a loss function such as hinge loss has a point where the derivative is not as easily computable. For example taking the derivative of the least square error and logistic loss we obtain,

$$
\begin{aligned}
L(\theta, \mathbf{y}) &= \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta)^2 \\
\frac{\partial}{\partial\hat{\theta}} L(\theta, \mathbf{y}) &= \frac{1}{n}\sum_{i=1}^{n} -2(y_i - \theta) \\
&= -\frac{2}{n}\sum_{i=1}^{n}(y_i - \theta)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial}{\partial\theta_j}\ell(\theta) &= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right)\frac{\partial}{\partial\theta_j} g(\theta^T x) \\
&= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right) g(\theta^T x)(1-g(\theta^T x))\frac{\partial}{\partial\theta_j}\theta^T x \\
&= \left(y(1-g(\theta^T x)) - (1-y)g(\theta^T x)\right) x_j \\
&= \left(y - h_\theta(x)\right) x_j
\end{aligned}
$$

Now the same procedure on the hinge loss gives us a different result with a piecewise derivative for the loss function as shown below.




In this case, SGD uses this piecewise gradient and since Hinge loss is convex we obtain convergence for this loss function.

# 9 Variants and Optimizations on Stochastic Gradient Descent

## 9.1 Introduction

Recall that the Gradient Descent update rule is

$$w^{t+1} = w^t - \eta \nabla_w \mathcal{L}(w^t),$$

where $w^t$ represents the weight vector at time step $t$ and $\eta$ is the learning rate hyperparameter that we set in advance. The gradient $\nabla_w \mathcal{L}(w^t)$ can be computed by taking an average over all data points or, as we have introduced, can be stochastically by computing the gradient contribution of an individual training example to the loss.

In the case of rather traditional linear and quadratic optimization problems, stochastic gradient descent works rather nicely, but when we are confronted with highly non-linear and nuanced losses (such as those in neural networks), getting stuck in local minima becomes a greater concern. In the subsequent sections, we will introduce several optimization strategies that allow us to both avoid local minima and also speed up optimization in scenarios where SGD is sub-optimal.

# 10 Momentum and Acceleration

## 10.1 Motivation

The analogy of a ball rolling down a hill is highly relevant when we think about optimizing losses with a complicated geometries. We can imagine the "ball" to be the current state of $w$, and depending on its current value, we update it to be a new value. Gradient Descent localizes this process; given the learning rate, the ball's new $w$ depends solely on its previous $w$, and the update is in accordance to the gradient at this $w$. Once $w$ is updated, it loses all memory of its previous state.

However, an actual ball rolling down a hill possesses some memory of its previous state - it is encoded in the speed by which it descends the hill. If a ball is travelling at a high speed, it is going moderately resistant to changes in its trajectory by new terrain and will sustain some of its previous motion. Naturally, the velocity from more recent times will play a more important role in the ball's motion, as opposed to velocities corresponding to times further in the past.

This actually perfectly sums up the idea of momentum in gradient descent: we will maintain an extra quantity $v$ as we descend, which encodes our descent velocity. Velocity should be updated in accordance to physical intuition - it should be a combination of the velocity at our previous time step (discounted by some amount) and the gradient of the loss function at our current weight $w^t$ (change in "terrain"). In the next section, we will formalize these ideas into actual update rules.

## 10.2 Momentum

The momentum update to the weight is

$$v^t = \gamma v^{t-1} + \eta \nabla_w \mathcal{L}(w^t),$$
$$w^t = w^{t+1} - v^t.$$

The parameter $\gamma$ is a discount factor we use to weigh velocities from previous time steps. If $\gamma$ equals 0, note that we are back to the regular stochastic gradient descent update rule because our update depends only on the gradient at $w^t$. The higher we make $\gamma$, the more "momentum" we have in our motion - the stronger the influence our previous update vector has on our current update vector.

Notice further that $v^{t-1}$ is dependent on $\gamma v^{t-2}$, which means that $v^t$ depends on $\gamma^2 v^{t-2}$ and generally, $v^t$ wil depend on $\gamma^k v^{t-k}$. So, even if $\gamma = 0.9$ (as it usually is in practice), after around 10 iterations of momentum based gradient descent, the influence of $v^{t-10}$ on $v^t$ will basically be negligible compared to the velocity at more recent time steps. This exponentially weighted update will be important for later optimizations.

## 10.3 Nesterov's Accelerated Gradient Descent

Perhaps one of the issues with momentum is with the fact that it can overshoot the optimum. A simple thought experiment to understand why this is an issue is in the case where the global optimum occurs in a very localized "well" in $w$-space surrounded by smooth terrain. While approaching the global optimum, it is possible for us to accumulate velocity to the point where we overshoot the global optimum. A more careful approach be more careful in taking the large steps that a large velocity entails.

The solution to the "overshooting" problem is to "look-ahead" as we descend. Nesterov's Accelerated Gradient Method does this by incorporating a simple twist on the Momentum Approach from the previous section:

$$v^t = \gamma v^{t-1} + \eta \nabla_w \mathcal{L}(w^t - \gamma v^{t-1}),$$
$$w^t = w^{t+1} - v^t.$$

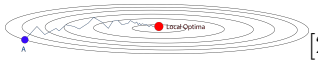
Instead of updating $v^t$ with the gradient at our current location $w^t$, we update it with the gradient at the point we would arrive at IF we took the momentum update $w^t - \gamma v^{t-1}$. This means that our current step depends on the gradient of where we expect to be on the next step (in addition to our previous velocity).

If we are close to a local minimum, then if we try to overshoot the minimum (as we would with naive momentum-based descent), the look-ahead gradient term will push us back, providing deceleration near the optimum. On the other hand, if the "look-ahead" gradient is points roughly in the same direction as our current velocity, we will get a "boost," or an acceleration in that direction.

# 11 Adaptive Learning Rate Methods

## 11.1 RMSProp and Motivation

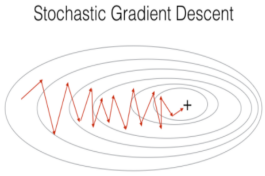
[3]




[2]

Figure 1: A scenario where it is more optimal for us to have a different learning rate in the horizontal and vertical directions.

View Figure 1. By the isocontours, it is clearly more optimal for us to have a high learning rate in the horizontal direction and a lower learning rate in the vertical direction (the optimization is very close to the optimum in the latter but not in the former). Standard stochastic gradient descent on this scenario can yield the picture shown in Figure 2 (that resembles a drunken walk)




[2]

Figure 2: A characterization on how SGD can perform on a loss function similar to the one shown in Figure 1.

Note that on each step that we both do not make much progress in the horizontal direction and that we have quite apparent oscillations in the vertical direction. The former will most definitely slow down convergence to the optimum, and the latter can cause us to not converge to the optimum exactly.

The question here is: can we adjust the learning rate to both dampen out the oscillations in the vertical direction and also speed up descent in the horizontal direction. The answer is yes and comes in the form of an algorithm called RMSProp.

The governing equations of RMS Prop are

$$w^t = w^{t-1} - \frac{\eta^t}{\sqrt{s^t + \epsilon}} \nabla_w \mathcal{L}(w^{t-1}),$$
$$s^t = \gamma s^{t-1} + (1-\gamma)(\nabla_w \mathcal{L}(w^{t-1}))^2.$$

As we progress through descent, we maintain a variable $s^t$, which represents an exponentially moving average of the norm squared of the gradient. (Note how this resembles the exponentially moving average of velocity that we did in momentum-based gradient descent.) Note further that we are taking an element-wise square root of $s^t + \epsilon$ and an element wise square of the gradient $\nabla_w \mathcal{L}(w^{t-1})$. The $\epsilon$ term is there to avoid numerical instability.

If our gradients have been large for a particular component in the past few time steps, then we will divide $\eta^t$ by a large number, implying more delicate updates in that component. This dampens out the oscillations in the vertical direction in Figure 2, for example.

If our gradients have been small for a particular component in the past few time steps, then we will divide $\eta^t$ by a small number, providing us a "boost" in that direction. This will speed up convergence in the horizontal direction in Figure 2.

We finally explore a method that synthesizes much of what we have discussed so far.

## 11.2 ADAM

[6]

ADAM stands for Adaptive Moment Estimation and is a state-of-the-art optimizer that is widely used for neural networks. ADAM, at its core, is just a combination of using momentum-based-descent with an adaptive learning rate (RMSProp). Much of the reasoning used for adaptive learning rates and momentum can be also utilized to justify ADAM - we want a velocity component to our descent to avoid getting stuck in local minima and a robust learning rate that is resistant to volatile oscillations and can progress delicately in some directions but aggressively in others. I will write the update rule for ADAM but will hold off on analysis, as you will be implementing ADAM and looking at its efficiency versus other methods in the coding assignment!

$$v^t = \beta_1 v^{t-1} + (1-\beta_1)\nabla_w \mathcal{L}(w^{t-1}),$$

$$s^t = \beta_2 s^{t-1} + (1-\beta_2)(\nabla_w \mathcal{L}(w^{t-1}))^2,$$

$$w^t = w^{t-1} - \frac{\eta v^t}{\sqrt{s^t + \epsilon}}.$$

There is actually something slightly wrong with this formula - the general idea is correct, but implementing this would not give a good optimizer. It is your job in the quiz questions to figure this out.

# References


[1] EECS 189 Fall 2020. *Optimization*. URL: https://www.eecs189.org/static/notes/n12.pdf.

[2] Data Science Deep Dive. *Optimizations of Gradient Descent*. URL: https://dsdeepdive.blogspot.com/2016/03/optimizations-of-gradient-descent.html?fbclid=IwAR2M6Qiaw-L9CEUFlKK5RR-L_CE95M3sL0vQjkBXFwplTpfvikrsFJr36vg.

[3] Muhammad Rizwan. *Gradient Descent with Momentum*. URL: https://engmrk.com/gradient-descent-with-momentum/?fbclid=IwAR39XQxIz4LYDy_XEozMDvSnYHKatKyYy1ZwWT8UTBUCozfCx5hkP7vVrtY.

[4] Muhammad Rizwan. *Mini-Batch Gradient Descent for Deep Learning*. URL: https://engmrk.com/mini-batch-gd/.

[5] Sklearn. *SKlearn SGDRegressor Documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html.

[6] Alex Smola. *L26/1 Momentum, Adagrad, RMSProp, Adam*. URL: https://www.youtube.com/watch?v=gmwxUy7NYpA.