

COMP 432 Machine Learning

Basic Machine Learning Concepts

Computer Science & Software Engineering
Concordia University, Fall 2024



Summary of the last episode....

- What is machine learning?
- Type of problems (*classification, regression*)
- Types of learning (*supervised, unsupervised*)
- Dataset and features
- Objective functions
- Parameters
- Training

Outline

- Capacity, Underfitting, Overfitting, Regularization
- Optimization
- Gradient Descent
- Hyperparameters
- Gradient Descent Variants

Basic Machine Learning Concepts:

Capacity

Capacity

- Different machine learning algorithms have different **hypothesis spaces**.
- The **capacity** (also called representational capacity) attempts to quantify how “*big*” (or “*rich*”) is the hypothesis space.



- It is usually not intended as the number of functions in the hypothesis space (that can easily be infinite also for simple machine learning models).
 - Instead, it more often refers to the **variability** in terms of “**family**” of **functions** (*expressive power, richness*).
-
- For instance, a complex model that can implement *linear*, *exponential*, *sinusoidal*, and *logarithmic* functions has a larger capacity than one that implements *linear* functions only.

VC Dimension

- One popular measure of the capacity of a model is the **Vapnik–Chervonenkis (VC) dimension**.
- It is defined as the **cardinality** of the **largest set of points** that a **binary classifier** can **shatter**.
- A set of points is **shattered** by the classifier if for all ways of splitting the examples into **positive** and **negative** subsets there exists a **perfect classifier**.
- If we have a set of **N** points $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N]^T$, we say that the **VC dimension** is **N** if at least one configuration of those data points can be shattered, but no set of **N+1** points can be shattered.

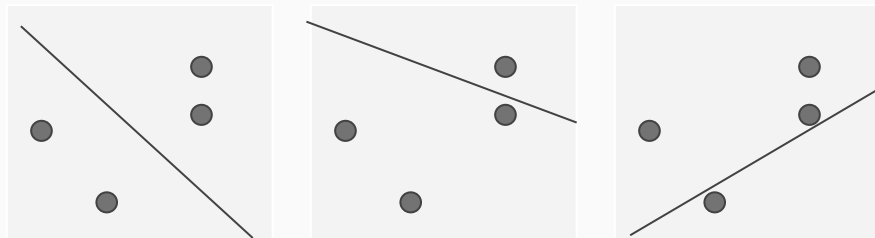
VC Dimension

Example: *Linear classifier* in a 2D space (binary)

$$f(x_1, x_2) = w_0 + w_1x_1 + w_2x_2$$

$$\hat{y} = \begin{cases} \text{class } 0, & \text{if } f(x_1, x_2) \geq 0 \\ \text{class } 1, & \text{otherwise} \end{cases}$$

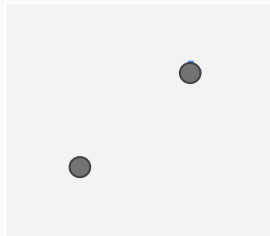
→ This classifier implements **linear boundaries**



- Let's now compute the VC dimension for this simple model.

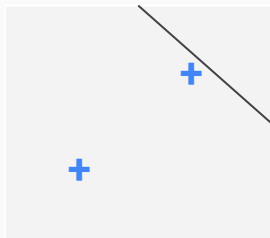
VC Dimension

- Let's start by drawing **2 random points**.

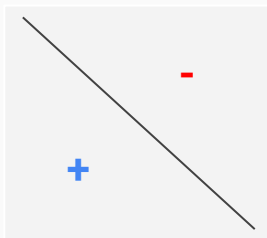


We need to assign random labels (positive and negative to these points)

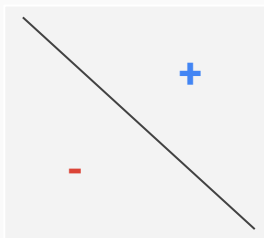
How many label configurations do we have? 4



yes!



yes!



yes!



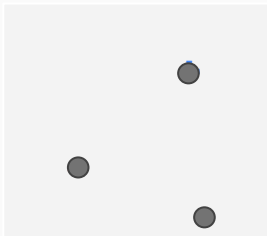
yes!

Can we draw a perfect classifier for all the label configurations?

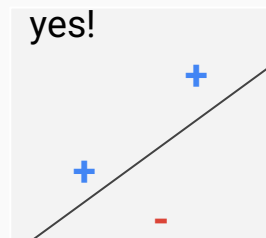
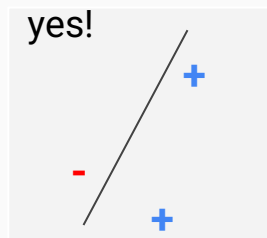
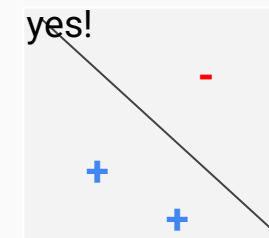
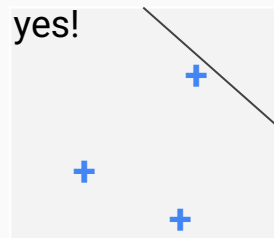
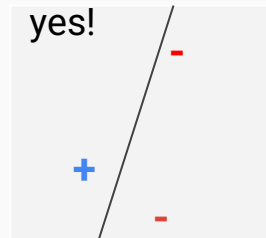
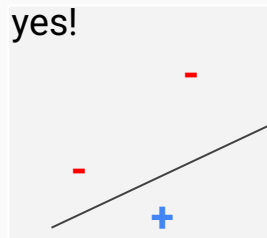
$$VC \geq 2$$

VC Dimension

- Let's draw **3 random points**!



How many label configurations do we have? 8

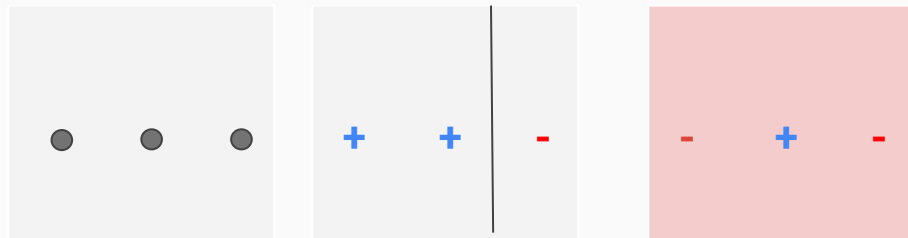


Can we draw a perfect classifier for all the label configurations?

$$VC \geq 3$$

VC Dimension

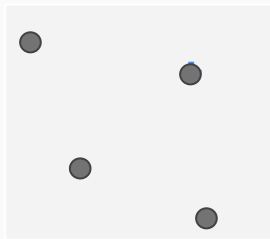
- Note that with some special point configurations, you cannot find a perfect classifier that works for all the label configurations.



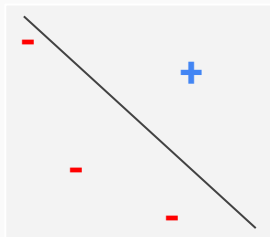
- We cannot classify correctly this set for all the label configurations.
- However, by definition, it is enough to find at **least one set of points** whose labels can be classified **correctly** with **all configurations**.

VC Dimension

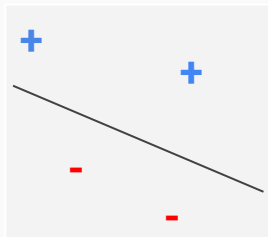
- Let's draw **4 random points**!



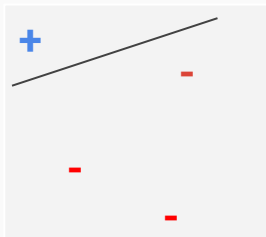
How many label configurations do we have? 16



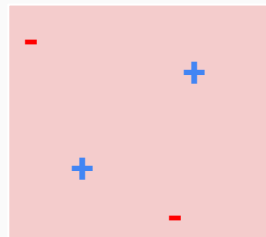
Yes



Yes



Yes



No

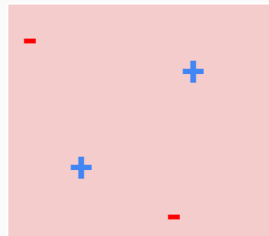
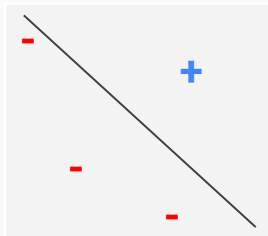
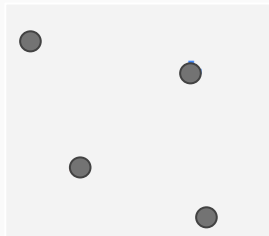
Can we draw a perfect classifier for all the label configurations?

$$VC = 3$$

- Try different points and you will see that you **cannot find any point configuration** that can lead to a perfect classifier for all the label configurations.

VC Dimension

4 points



- The 4 points cannot be shattered by this linear classifier:

$$VC = 3$$

- It can be shown that for a binary linear classifier operating on D-dimensional inputs:

$$f(x_1, x_2, \dots, x_D) = w_0 + \sum_{i=1}^D w_i x_i$$

$$VC = D + 1$$

- The VC dimension thus depends on “how” complex the classifier can be.
- The larger D, the larger the set of parameters of the classifier.

Basic Machine Learning Concepts:

Generalization, Overfitting, Underfitting

Generalization

- Training a machine learning model often requires solving an **optimization problem**.

$$\theta^* = \operatorname{argmin}_{\theta} J(\mathbf{Y}, f(\mathbf{X}, \theta))$$

We have to find the parameters of the function f that minimizes the objective function using the **training data**.

- However, we are more interested in the performance achieved on data **never seen before**.
- **Generalization** is the ability of a machine learning algorithm to perform well on new, previously unseen data.
- A machine learning model generalizes well if the test loss is low enough (according to our application)

Training Loss: Objective function computed with the training set.

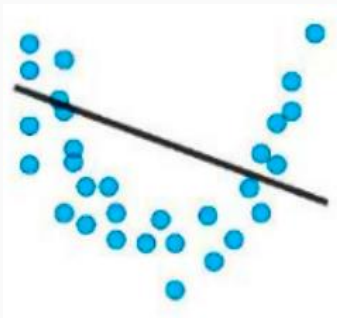
Test Loss: Objective function computed with the test set.

Underfitting

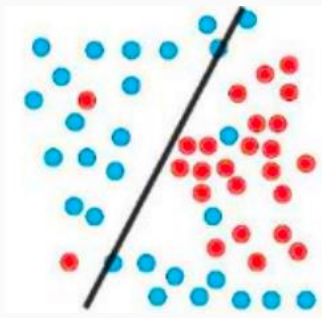
- By analyzing training and test losses, we can identify some “**pathologies**” that often affect machine learning models.



- One of these conditions is called **underfitting**.
- It happens when the model cannot achieve a **training loss** sufficiently **low**.



Regression



Classification

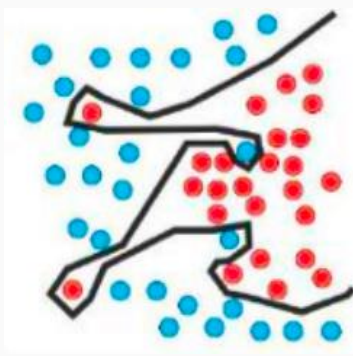
- In both cases, the function found by the machine learning algorithm is **too simple** to explain well the training data.

Overfitting

- Another pathology is **overfitting**.
- It happens when the **gap between the training and test losses** is too **large**.



Regression



Classification

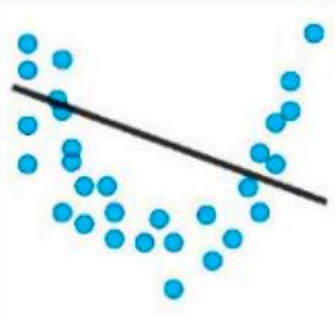
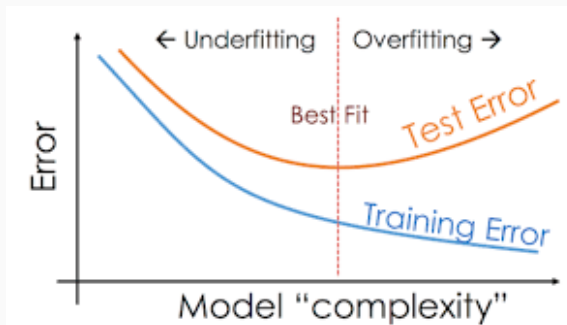
- In both cases, the learned function is **too complex** to explain well the training data.
- In an extreme case, the model stores the training samples and just behaves as a **memory without generalization capabilities**.

Underfitting, Overfitting, Capacity

- **Underfitting** and **Overfitting** are connected to the **capacity** of the model.

- Intuitively:

Too Low Capacity → Underfitting
Proper Capacity → Proper Fitting
Too High Capacity → Overfitting



Underfitting
(low capacity)



Proper Fitting
(proper capacity)



Overfitting
(low capacity)

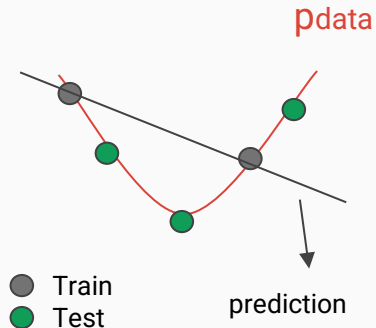
For each task, we have to choose a model with **proper complexity**.

Underfitting, Overfitting, Dataset size

- **Underfitting** and **Overfitting** are influenced by the **number of training samples** as well.
- Intuitively:

More Training Examples → Better Generalization
Few Training Examples → Overfitting

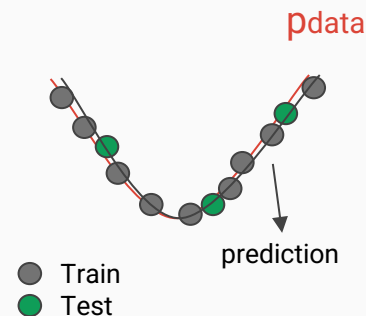
Small Number of Training Samples



In this case, the **training loss** is **low**.

We are anyway in an **overfitting** regime because the **test loss** is **high**.

High number of Training Samples

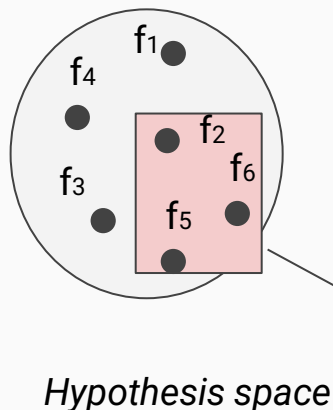


Also in this case the **training loss** is **low**.

However, this time we have a good generalization because **test loss** is **low**.

Regularization

- **Regularization** techniques aim to counteract overfitting (and thus improve generalization).
- Different techniques have been proposed in the machine learning literature (e.g., L1 regularization, L2 regularization).
- One way to regularize is to express some **preference for some solutions** over others (using **prior knowledge**).



- For instance, we can **penalize complexity**.



Occam's razor (1287-1346)

Among competing hypotheses that explain the training data equally well, we should choose the "simplest" one

This selection is based on prior knowledge on how could be a proper function for my problem.

COMP 432 Machine Learning

Basic Machine Learning Concepts: Optimization

Computer Science & Software Engineering
Concordia University, Winter 2022



Optimization

- Training a machine learning model often requires solving an **optimization problem**.

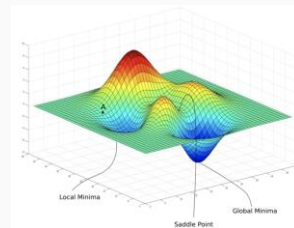
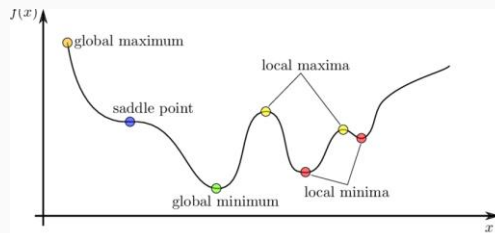
For *parametric machine learning*:

$$\theta^* = \operatorname{argmin}_{\theta} J(\mathbf{Y}, f(\mathbf{X}, \theta))$$

We want to find the parameters of the function f that minimizes the objective function.

This is usually a **difficult** problem:

- The objective function might have **local minima, maxima, and saddle points**.
- In a real machine learning problem, we might have **tons of parameters to optimize**.
- For instance, in modern deep learning, we might even have **billions of parameters**.



Critical Points

Single Parameter case

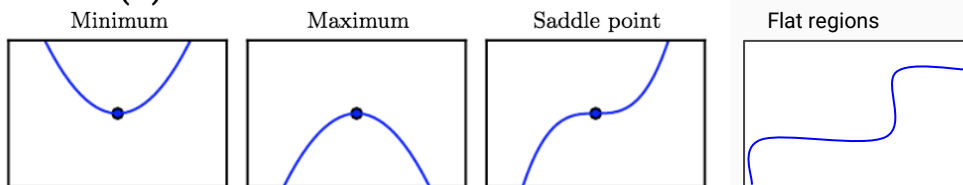
- **Critical points:** points where $\frac{\partial J(\theta)}{\partial \theta} = 0$
- **Local Minimum:** it is a critical point where $J(\theta)$ is lower than all the neighboring points.
- **Local Maximum:** it is a critical point where $J(\theta)$ is higher than all the neighboring points.
- **Saddle point:** A critical point that is not a maximum nor a minimum value.
- **Plateau:** wide areas where the value of $J(\theta)$ is constant.

Derivative

$$\frac{\partial J(\theta)}{\partial \theta} = \lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon}$$

If this limit exists for any value of θ , the function is said **differentiable**.

The derivative tells us how the objective function J changes with a little change in the parameters θ .



Gradient

Multiple Parameters

- When we have more than one parameter we have to compute the **partial derivatives** over all the parameters:

$$\boldsymbol{\theta} = [\theta_1, \theta_2, \dots, \theta_M]^T \quad \text{Partial Derivatives}$$

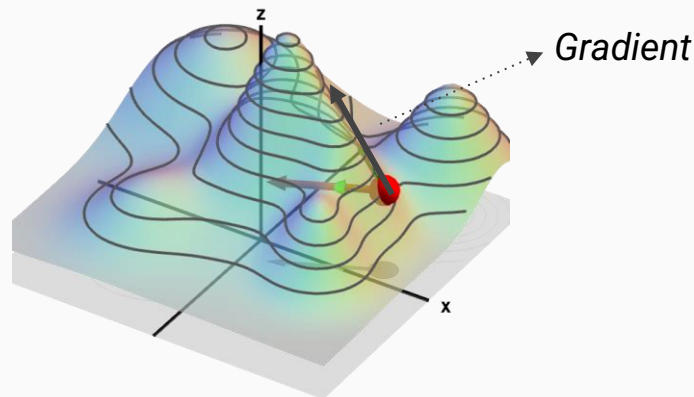
$$\nabla J(\boldsymbol{\theta}) = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \dots, \frac{\partial J}{\partial \theta_M} \right]^T$$

Gradient

How J changes when I apply a little perturbation to θ_1 ?

How J changes when I apply a little perturbation to θ_M ?

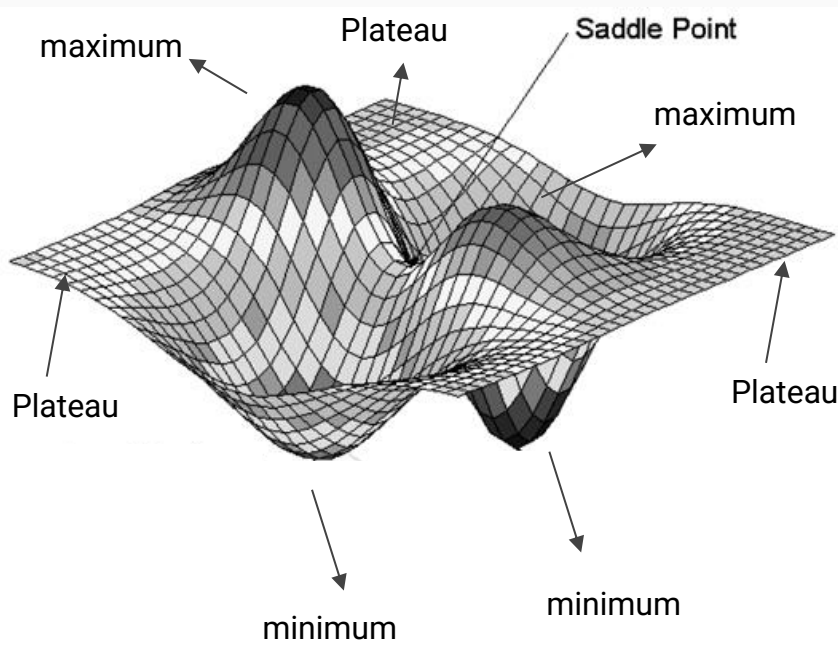
- The **gradient** generalizes the notion of derivative.
- It is a **vector** that points in the direction of the **greatest increase** of the objective.



Critical Points in Multi-dimensional Spaces

Multiple Parameters

- **Critical points:** points where $\nabla J(\boldsymbol{\theta}) = 0$



- In high-dimensional spaces, there is a **proliferation** of local *minima*, *maxima*, and *saddle points*.
- In particular, there is a proliferation of *saddle points*.
- Saddle points are a **local minimum** along one dimension (or cross-section of the objective function) and a **local maximum** in another one.
- The ratio between the expected number of saddle points and the local minima **grows exponentially** with dimensionality.
- This makes the optimization problem **very challenging**.

How can we solve the optimization problem?

Analytical Solution

- How can we solve the optimization problem?

$$\theta^* = \operatorname{argmin}_{\theta} J(\mathbf{Y}, f(\mathbf{X}, \theta))$$

We can try to find an **analytical solution** to the problem:

1. We have to compute an expression for the **gradient** $\nabla J(\theta)$
2. We find a **closed-form expression** that tells us where are all the **critical points**: $\nabla J(\theta) = 0$
3. We test **all the critical points** and we choose the one that minimizes the objective J.



Most of the time, we **cannot find** a **closed-form analytical expression** that solves this problem (*root finding*).

Closed-form expressions only exist for **simple functions** (e.g, *linear*).

Numerical Optimization

- How can we solve the optimization problem?

$$\theta^* = \operatorname{argmin}_{\theta} J(\mathbf{Y}, f(\mathbf{X}, \theta))$$

Fortunately, we can try to find a **numerical solution** to this problem.

- With this approach, we accept the idea of finding an **approximate solution** to the problem.
- When using numerical methods, we start from a set of candidate solutions, and we **progressively improve** them until we think the problem is solved **well enough**.

Numerical Optimization

Naive Approach

$$\theta^* = \underset{\theta}{\operatorname{argmin}} J(\mathbf{Y}, f(\mathbf{X}, \theta))$$



Idea 1: *We can try all the parameters θ and choose the set that optimizes the objective J*



Very often the parameters are continuous and we have an **infinite number** of **parameter configurations**.



Idea 2: *We can (randomly) sample N parameter configurations and take the set that optimizes the objective J*



The number of parameter combinations grows **exponentially** with the number of parameters.

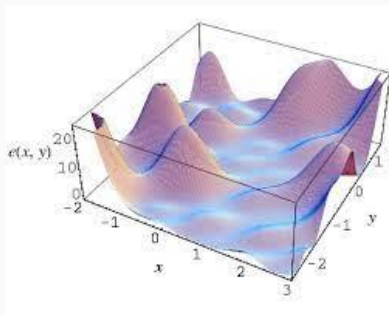
In a high-dimensional space, the **probability** of finding a good set of parameters by random sampling is **almost zero**.



What's the probability of a monkey typing Shakespeare?

Numerical Optimization

- Numerical Optimization is a big research field and many approaches have been proposed so far.
- Some attempts to find a **global optimum** (e.g., *simulated annealing, ant colony, genetic algorithms, particle swarm optimization*). This is usually **computationally demanding**.
- Some of them can find **local optima** only (e.g., *hill-climbing, coordinate descent, gradient descent, conjugate gradient method, Newton's Methods*).

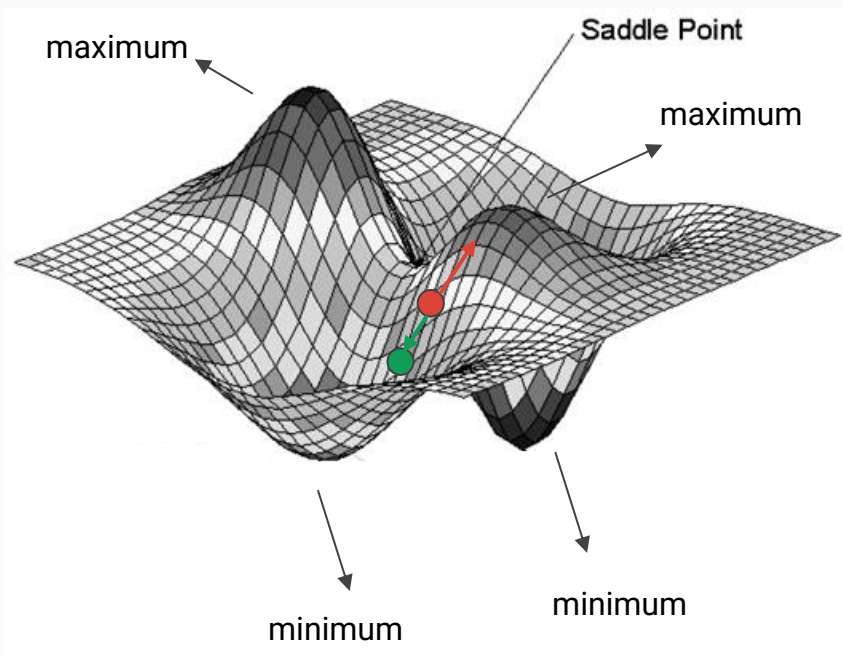


- In this case, we accept a “good enough” **sub-optimal solution**.
- Local optimization is **much faster** than global optimization.
- It is thus the **dominant approach** in current machine learning.

Gradient Descent

- We have seen that the gradient $\nabla J(\theta)$ gives us useful information:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} J(\mathbf{Y}, f(\mathbf{X}, \theta))$$



The gradient points in the direction of the **greatest increase** of the objective.

If we want to minimize J , we might want to do a **little step** in this direction:

$$\theta_{new} = \theta - \eta \nabla J(\theta)$$

New
parameters

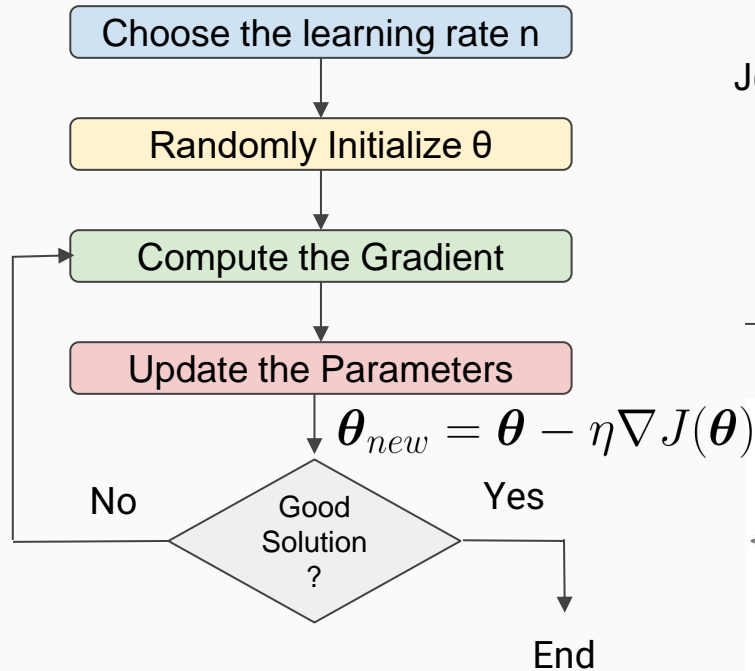
Current
parameters

Learning Rate

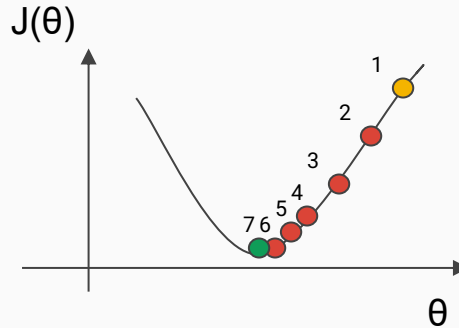
Gradient

Gradient Descent

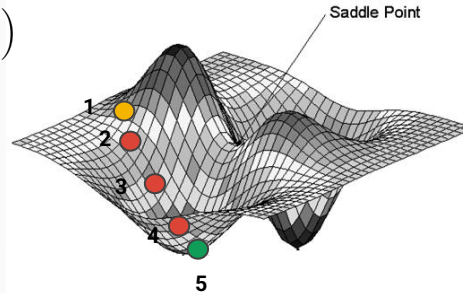
Algorithm



$$\theta^* = \underset{\theta}{\operatorname{argmin}} J(\mathbf{Y}, f(\mathbf{X}, \theta))$$



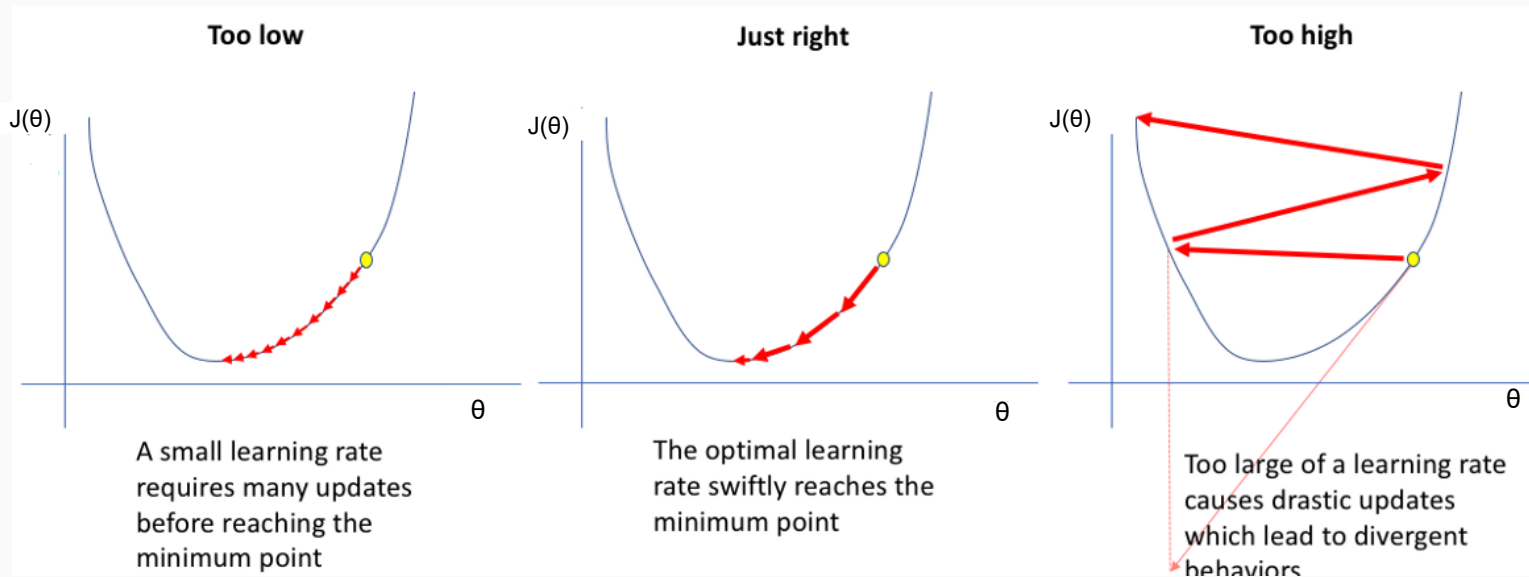
We start from a **random solution** and we progressively improve it until convergence.



The algorithm is **sensitive** to the **initialization** point.

Learning Rate

- Setting a proper **learning rate** ϵ is crucial for gradient descent. A common choice is to set it as a constant value (e.g., $\epsilon = 0.01$).



Stochastic Gradient Descent

- To improve generalization, modern machine learning models are trained on **large datasets**.
- In standard gradient descent, we update the parameters based on the **gradient**.
- To compute the gradient, we need to process all the training examples:

$$J(\mathbf{Y}, f(\mathbf{X}), \boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N L(f(\mathbf{x}_i, \boldsymbol{\theta}), \mathbf{y}_i)$$

The objective function is often decomposed as a **sum over training samples** (empirical risk minimization).

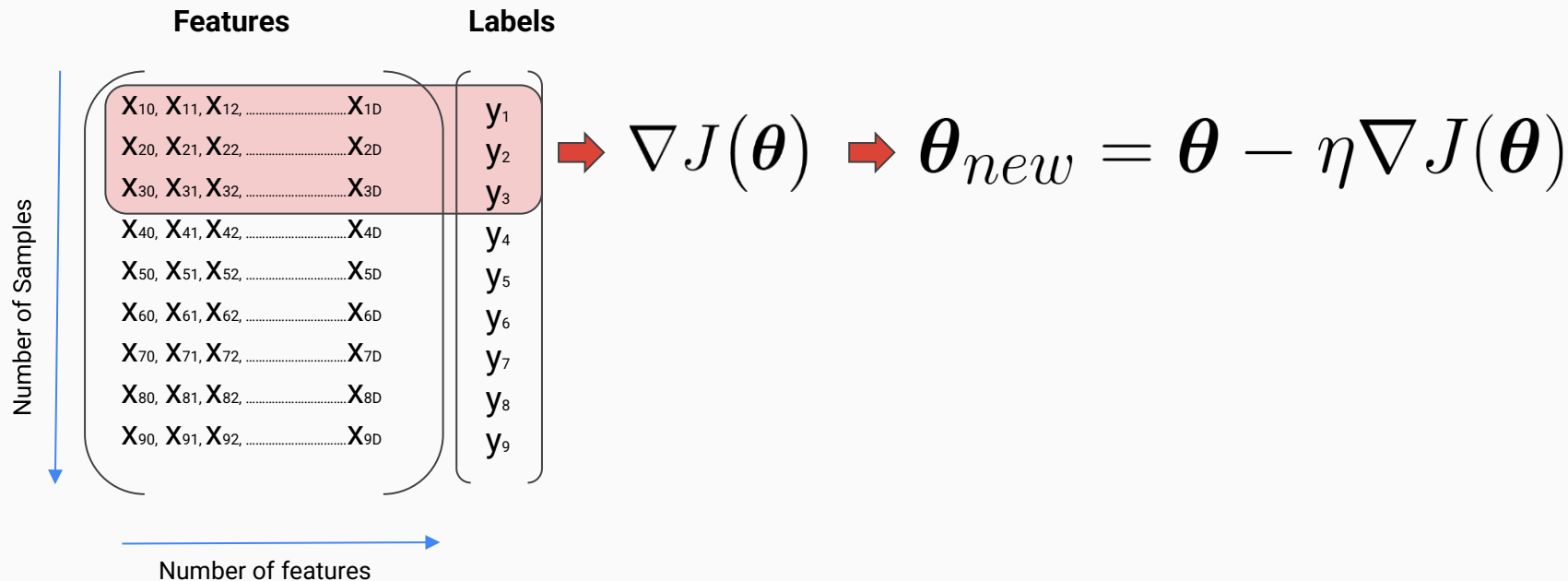
$$\nabla J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \nabla L_i(\boldsymbol{\theta})$$

The total gradient is the average of the gradients computed for **each training sample**.

The complexity of gradient computation is $O(N)$.

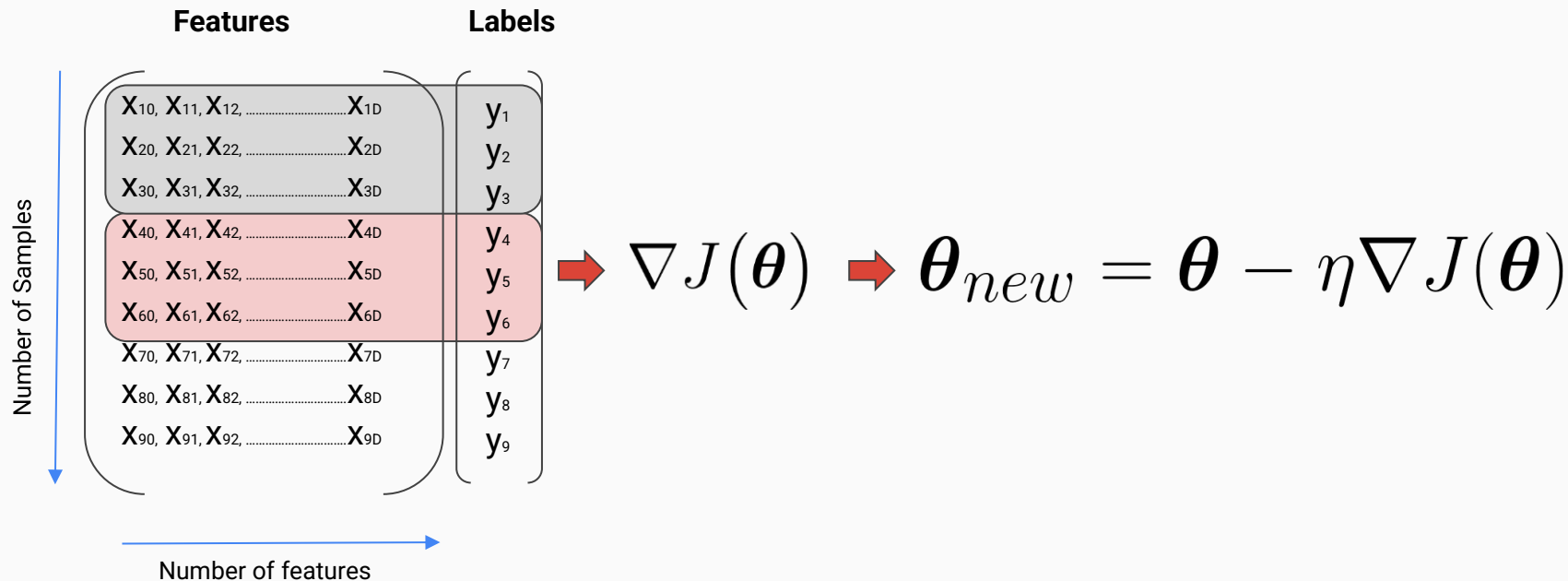
Stochastic Gradient Descent

- What about approximating the gradient using little training data?



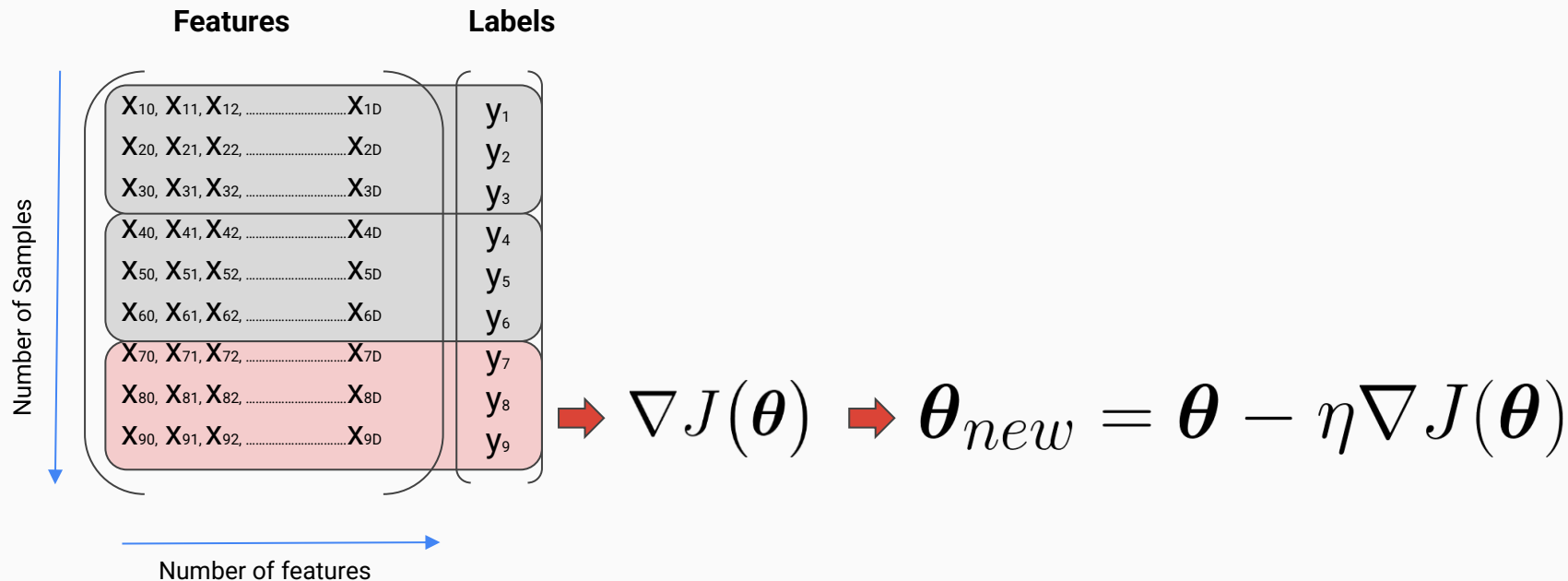
Stochastic Gradient Descent

- What about approximating the gradient using little training data?



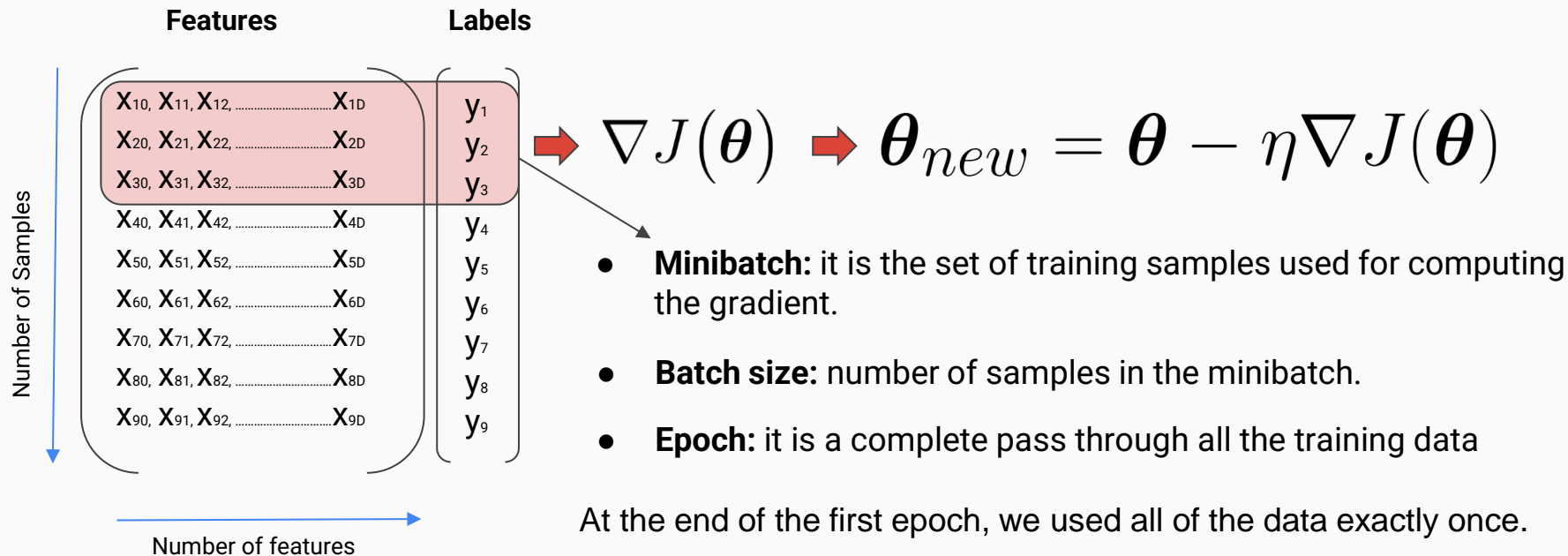
Stochastic Gradient Descent

- What about approximating the gradient using little training data?



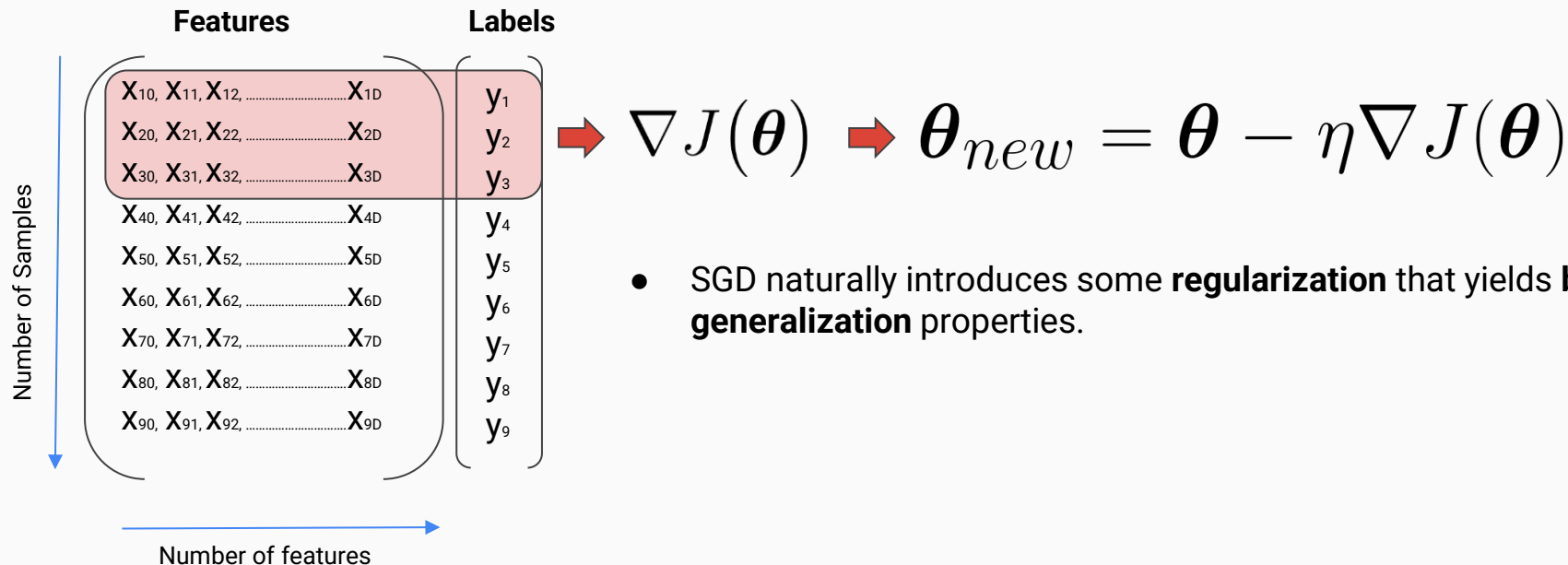
Stochastic Gradient Descent

- What about approximating the gradient using little training data?



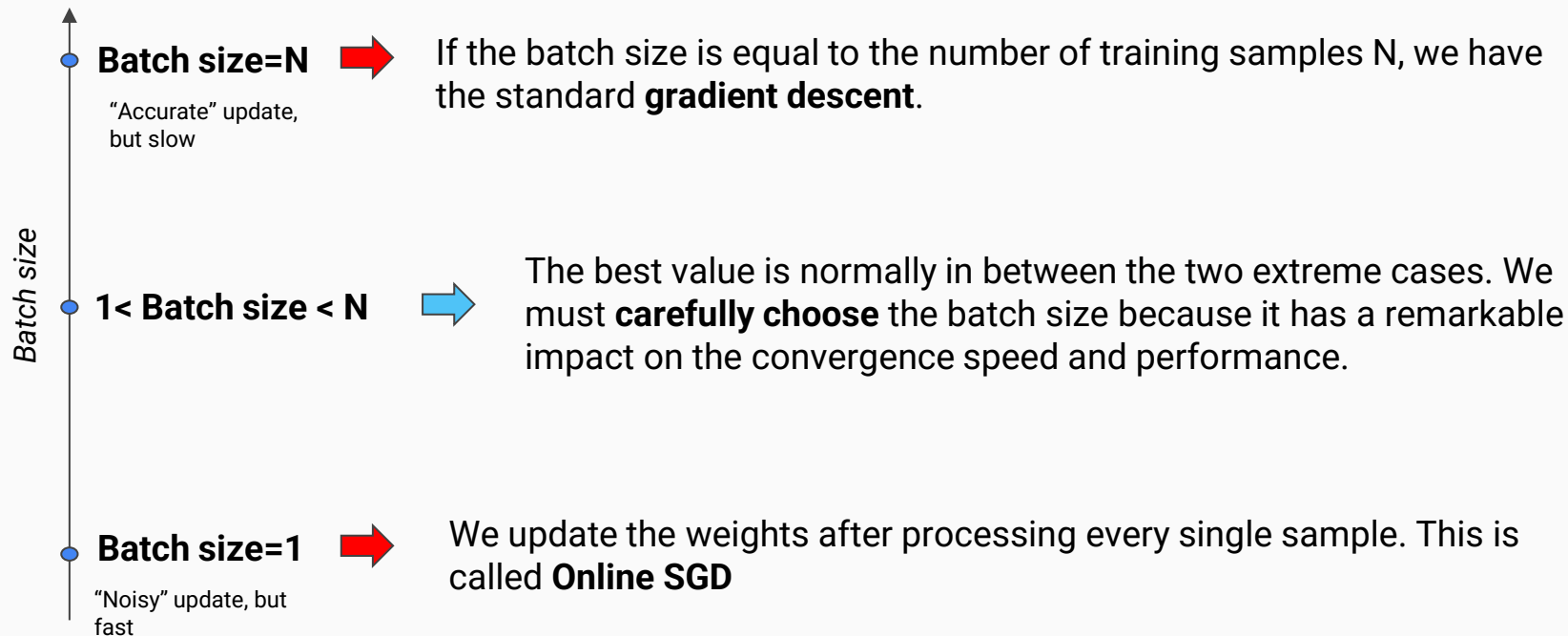
Stochastic Gradient Descent

- With SGD we use **noisy versions** of the gradient.
- This introduces some **randomness** in the learning process that helps the algorithm to **escape** from **saddle points** and **local minima**.



Batch Size

- The **batch size** manages the trade-off between the “*accuracy*” and the *computational cost* of the updates.



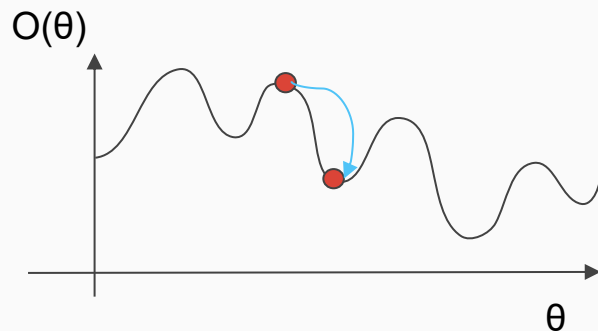
Advantages and Limitations

Limitations

- ☹️ Gradient descent can get stuck in local optima.
- ☹️ It depends on proper initialization.
- ☹️ It requires tuning the learning rate (and batch size for SGD).
- ☹️ It is difficult to apply to non-differentiable loss functions.

Advantages

- 😊 Computationally efficient (computing the gradient is fast usually).
- 😊 It works well in practice (even with nonconvex objectives).
e.g., It proved capable of escaping from saddle points.



From the “Deep Learning Book”

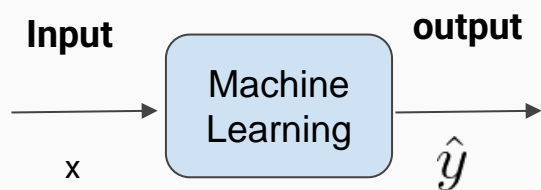
“In the past the application of gradient descent to nonconvex optimization problems was regarded as foolhardy or unprincipled. Today, we know that machine learning models work very well when trained with gradient descent.”

“The optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but often finds a very low value of the cost function quickly enough to be useful”

Example: Linear Least Square

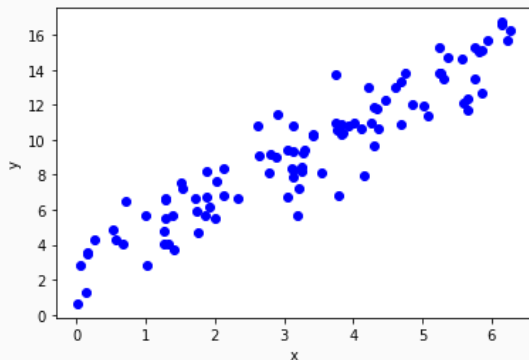
Example - Linear Least Squares

- Let's try to apply gradient descent in a simple **linear regression** problem.



$$\hat{y} = f(x, \boldsymbol{\theta}) \Rightarrow \hat{y} = w_0 + w_1 x$$
$$\boldsymbol{\theta} = \mathbf{w} = [w_0, w_1]^T$$

Training set



Inputs

$$\mathbf{X} = [x_1, x_2, \dots, x_N]^T$$

Labels

$$\mathbf{Y} = [y_1, y_2, \dots, y_N]^T$$

Objective (MSE)

$$J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^N (y_i - f(x_i, \mathbf{w}))^2$$

$\hat{y}_i = w_0 + w_1 x_i$

An upward-pointing arrow connects the predicted value \hat{y}_i in the equation below to the $f(x_i, \mathbf{w})$ term in the equation above.

Example - Linear Least Squares

- To apply gradient descent, we need to **compute the gradient** of the objective function J

$$\begin{aligned} J(\mathbf{w}) &= \frac{1}{2} \sum_{i=1}^N (y_i - f(x_i, \mathbf{w}))^2 \\ &= \frac{1}{2} \sum_{i=1}^N (y_i - w_0 - w_1 x_i)^2 \end{aligned}$$

$$\nabla J(\boldsymbol{\theta}) = \left[\frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1} \right]$$

$$\begin{aligned} \frac{\partial J}{\partial w_0} &= 2 \cdot \frac{1}{2} \sum_{i=1}^N (y_i - \hat{y}_i) \cdot (-1) \\ &= \sum_{i=1}^N (\hat{y}_i - y_i) \end{aligned}$$

Example - Linear Least Squares

- To apply gradient descent, we need to **compute the gradient** of the objective function J

$$\begin{aligned} J(\mathbf{w}) &= \frac{1}{2} \sum_{i=1}^N (y_i - f(x_i, \mathbf{w}))^2 \\ &= \frac{1}{2} \sum_{i=1}^N (y_i - w_0 - w_1 x_i)^2 \end{aligned}$$

$$\nabla J(\boldsymbol{\theta}) = \left[\frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1} \right]$$

$$\begin{aligned} \frac{\partial J}{\partial w_1} &= 2 \cdot \frac{1}{2} \sum_{i=1}^N (y_i - \hat{y}_i) \cdot (-x_i) \\ &= \sum_{i=1}^N (\hat{y}_i - y_i) x_i \end{aligned}$$

Example - Linear Least Squares

- Now that we have the gradient, we can start the gradient descend training:

```
# Initial Values
w0 = 1.0
w1 = 0.5

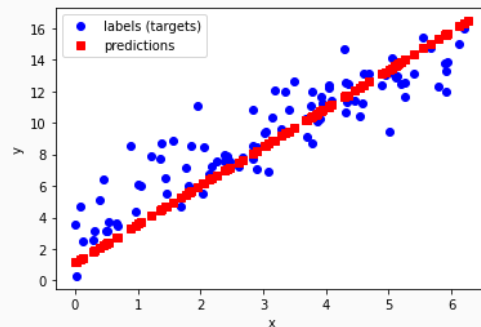
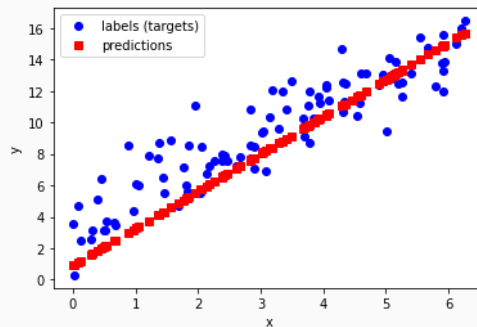
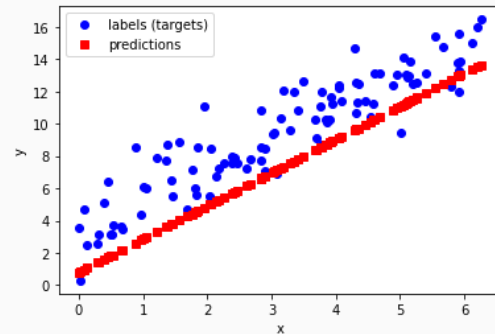
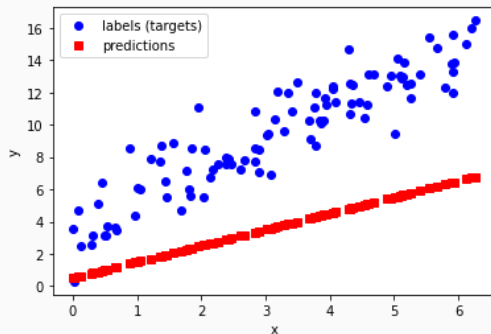
N_epochs = 10
lr = 0.05

train_loss = []
test_loss = []

for epoch in range(N_epochs):
    # compute the predictions
    y_hat = linear_model(x_train, w0, w1)

    # compute the gradient
    grad_w0 = (y_hat - y_train).mean()
    grad_w1 = ((y_hat - y_train) * x_train).mean()

    # parameter updates
    w0 = w0 - lr * grad_w0
    w1 = w1 - lr * grad_w1
```



Example - Linear Least Squares

- Now that we have the gradient, we can start the gradient descend training:

```
# Initial Values
w0 = 1.0
w1 = 0.5

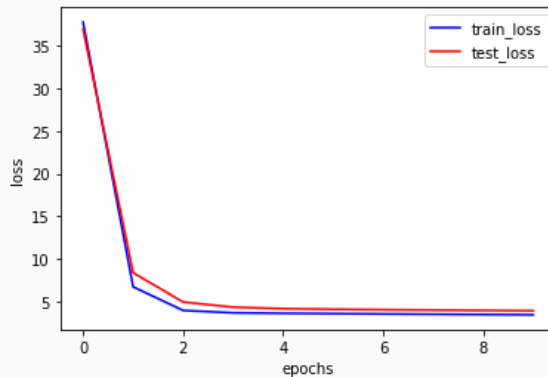
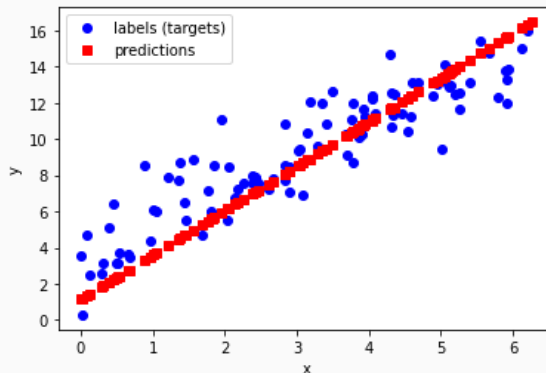
N_epochs = 10
lr = 0.05

train_loss = []
test_loss = []

for epoch in range(N_epochs):
    # compute the predictions
    y_hat = linear_model(x_train, w0, w1)

    # compute the gradient
    grad_w0 = (y_hat - y_train).mean()
    grad_w1 = ((y_hat - y_train) * x_train).mean()

    # parameter updates
    w0 = w0 - lr * grad_w0
    w1 = w1 - lr * grad_w1
```



Example - Linear Least Squares

- **Note:** Sometimes it is convenient to write the linear model in a vector form:

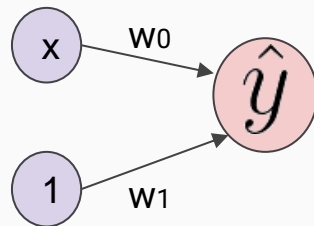
1D Input

$$\hat{y} = \mathbf{X}^T \mathbf{W} = \mathbf{W}^T \mathbf{X}$$

$$\mathbf{X} = [\textcircled{1}, x]^T \quad \mathbf{W} = [w_0, w_1]^T$$

Appending this one is needed to manage the intercept term w_0

$$\begin{bmatrix} 1 & x \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = w_0 + w_1 x$$

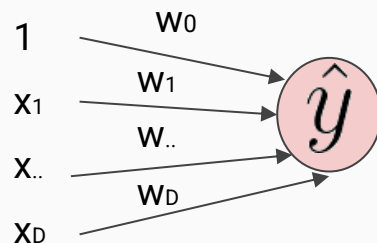


Example - Linear Least Squares

- **Note:** Sometimes it is convenient to write the linear model in a vector form:

Multiple D-dimensional Input

$$\hat{y} = \mathbf{X}\mathbf{w}$$



Needed to vectorize the intercept terms

$$\begin{bmatrix} 1 & x_{11} & x_{11} & \dots & x_{1D} \\ 1 & x_{21} & x_{21} & \dots & x_{2D} \\ 1 & \dots & \dots & \dots & \dots \\ 1 & x_{N1} & x_{N1} & \dots & x_{ND} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_D \end{bmatrix} = \begin{bmatrix} w_0 + w_1x_{11} + w_2x_{12} + \dots + w_Dx_{1D} \\ w_0 + w_1x_{21} + w_2x_{22} + \dots + w_Dx_{2D} \\ \dots \\ w_0 + w_1x_{N1} + w_2x_{N2} + \dots + w_Dx_{ND} \end{bmatrix}$$

The number of parameters $P = D + 1$ for this model (D is the feature dim) 48

Example - Linear Least Squares

- In the multidimensional case, we can generalize the gradient as:

$$\nabla J(\mathbf{w}) = \begin{bmatrix} \frac{\partial J(\mathbf{w})}{\partial w_0} \\ \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \dots \\ \frac{\partial J(\mathbf{w})}{\partial w_D} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w} - y_i) \\ \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w} - y_i) x_{i1} \\ \dots \\ \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w} - y_i) x_{iD} \end{bmatrix} = \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w} - y_i) \mathbf{x}_i$$

$$\mathbf{x}_i = [1, x_0, \dots, x_D]^T \quad \mathbf{w} = [w_0, w_1, \dots, w_D]^T$$

- If we set:

$$\mathbf{X} = [1, x]^T \quad \mathbf{w} = [w_0, w_1]^T \quad \rightarrow \quad \text{We obtain the equations seen for the 1d case}$$

Example - Linear Least Squares

- We can also write the gradient equations in **matrix form**:

$$\nabla J(\mathbf{w}) = \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w} - y_i) \mathbf{x}_i = (\mathbf{X}^T \mathbf{X}) \mathbf{w} - \mathbf{X}^T \mathbf{y}$$

$$\mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & \dots & x_{1,D} \\ 1 & x_{2,1} & \dots & x_{2,D} \\ 1 & \dots & \dots & \dots \\ 1 & x_{N,1} & \dots & x_{N,D} \end{bmatrix} \quad \mathbf{w} = [w_0, w_1, \dots, w_D]^T$$
$$\mathbf{y} = [y_1, y_2, \dots, y_N]^T$$

Example - Linear Least Squares

- Let's do a **sanity check** on the dimensionalities to convince us better than the two expressions are equivalent.

$$\nabla J(\mathbf{w}) = \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w} - y_i) \mathbf{x}_i = (\mathbf{X}^T \mathbf{X}) \mathbf{w} - \mathbf{X}^T \mathbf{y}$$

- Let's assume we have 2 parameters w_0 , and w_1 .
- What is the expected dimensionality for the gradient? (2 x 1)
- What is the dimensionality of \mathbf{w} ? (2 x 1)
- Let's assume we have 3 examples. What will be the dimensionality of the feature matrix \mathbf{X} ? (3 x 2)


Example - Linear Least Squares

- Let's do a **sanity check** on the dimensionalities to convince us better than the two expressions are equivalent.

$$\nabla J(\mathbf{w}) = \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w} - y_i) \mathbf{x}_i = \underbrace{(\underbrace{\mathbf{X}^T}_{(2,3)} \underbrace{\mathbf{X}}_{(3,2)})}_{(2,2)} \mathbf{w} - \underbrace{\mathbf{X}^T}_{(2,3)} \underbrace{\mathbf{y}}_{(3,1)}_{(2,1)}$$

Why do we want expressions that contain matrices?

- For the sake of compactness.
- We can take advantage of fast matrix multiplication libraries.

 This is the expected dimensionality for the gradient.

→ (2,1)

- If you expand the computations for both expressions, you will see the exact same operations (do it as an additional exercise).

Example - Linear Least Squares

This problem is simple enough and can be solved in a **closed form** with an **analytical expression**:

$$\nabla J(\mathbf{w}) = (\mathbf{X}^T \mathbf{X}) \mathbf{w} - \mathbf{X}^T \mathbf{y} = 0$$

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Moore-Penrose Pseudoinverse



Generalization of the notion of “inverse” matrix to non-square matrices

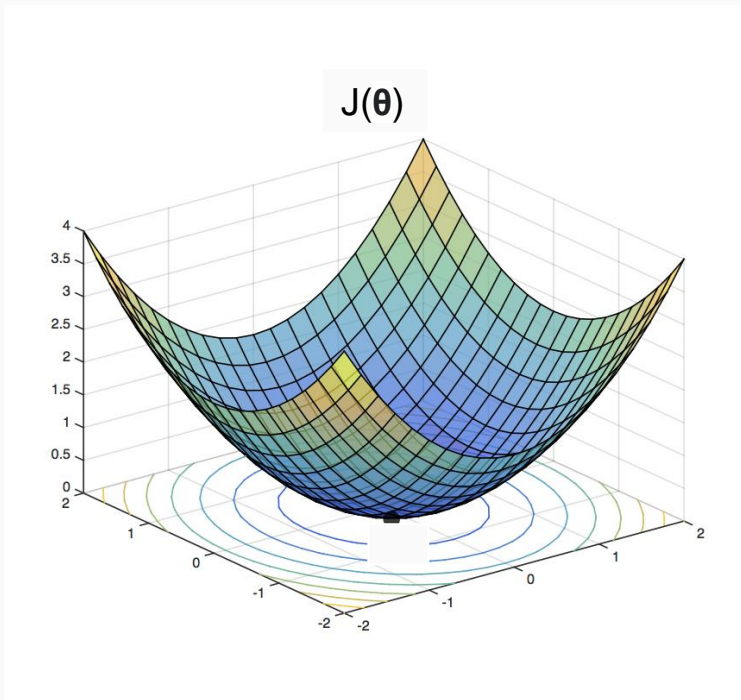
We can solve this simple problem in one shot (without using gradient descent) just by solving a system of linear equations.



This is possible only for such a simple machine learning model. For more complex ones, the closed-form solution does not exist.

Example - Linear Least Squares

For this kind of simple model (linear model trained with MSE), it can be shown that the objective function is **convex**:

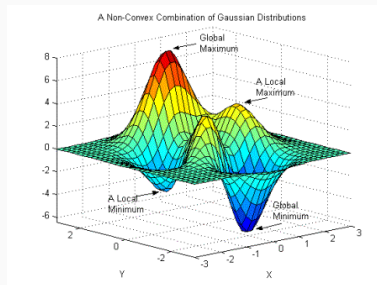
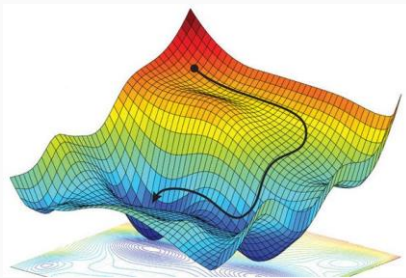


- The function has only one minimum which is the global one.



This happens only in this simple case.

More often, we have to solve non-convex optimization problems.



Example - Linear Least Squares

`sklearn.linear_model.LinearRegression`

```
class sklearn.linear_model.LinearRegression(*, fit_intercept=True, normalize='deprecated', copy_X=True, n_jobs=None,
positive=False) \[source\]
```

Ordinary least squares Linear Regression.

`LinearRegression` fits a linear model with coefficients $w = (w_1, \dots, w_p)$ to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

Parameters: `fit_intercept : bool, default=True`

Whether to calculate the intercept for this model. If set to `False`, no intercept will be used in calculations (i.e. data is expected to be centered).

`normalize : bool, default=False`

This parameter is ignored when `fit_intercept` is set to `False`. If `True`, the regressors `X` will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use `StandardScaler` before calling `fit` on an estimator with `normalize=False`.

Deprecated since version 1.0: `normalize` was deprecated in version 1.0 and will be removed in 1.2.

`copy_X : bool, default=True`

If `True`, `X` will be copied; else, it may be overwritten.

`n_jobs : int, default=None`

The number of jobs to use for the computation. This will only provide speedup in case of sufficiently large problems, that is if firstly `n_targets > 1` and secondly `X` is sparse or if `positive` is set to `True`. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See [Glossary](#) for more details.

`positive : bool, default=False`

When set to `True`, forces the coefficients to be positive. This option is only supported for dense arrays.

New in version 0.24.

Hyperparameters and Validation Set

Hyperparameters

- Several machine learning models have to **learn** the **parameters θ** that implement the desired input-output mapping (e.g., the slope w_1 and the intercept w_0 of our linear model)
- There are special “parameters” to set to properly **control the learning algorithm itself**.
- These “special parameters” are called **hyperparameters**.
- In the context of SGD, the **learning rate**, the **batch size**, and the **number of epochs** are examples of hyperparameters.
- We cannot compute the gradient for these variables and users have to set them manually.

Hyperparameters



How do we choose the hyperparameters?

- One way is to perform **several training experiments** with different sets of hyperparameters and choose the **best one**.



To select the best one, **we cannot use** the performance achieved on the **training set**.

Why?

Because we increase the risk of overfitting.



To select the best one, **we cannot use** the performance achieved on the **test set**.

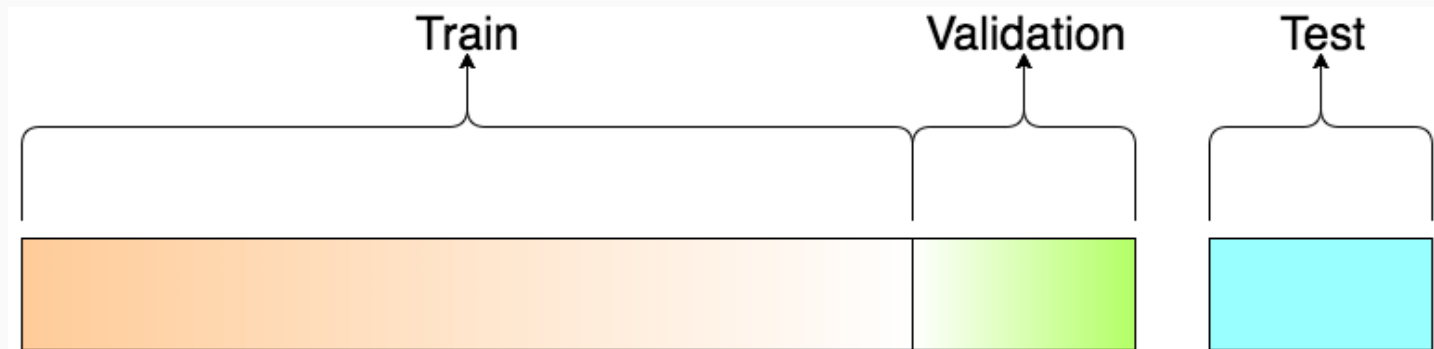
Why?

Because we will overestimate the actual performance of the system.



Validation Set

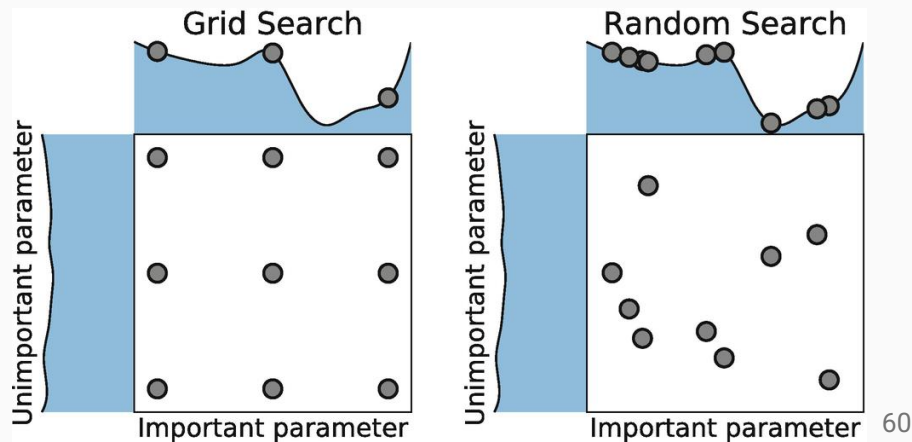
- We can employ a third set, called **validation set**, to choose the best set of hyperparameters.
- The validation set is normally extracted from the training set (e.g, 10%-20% of the training data are devoted to validation).
- The **training set** is used to find the **best parameters**, the **validation set** is used only to select the **hyperparameters**.



Hyperparameter Search

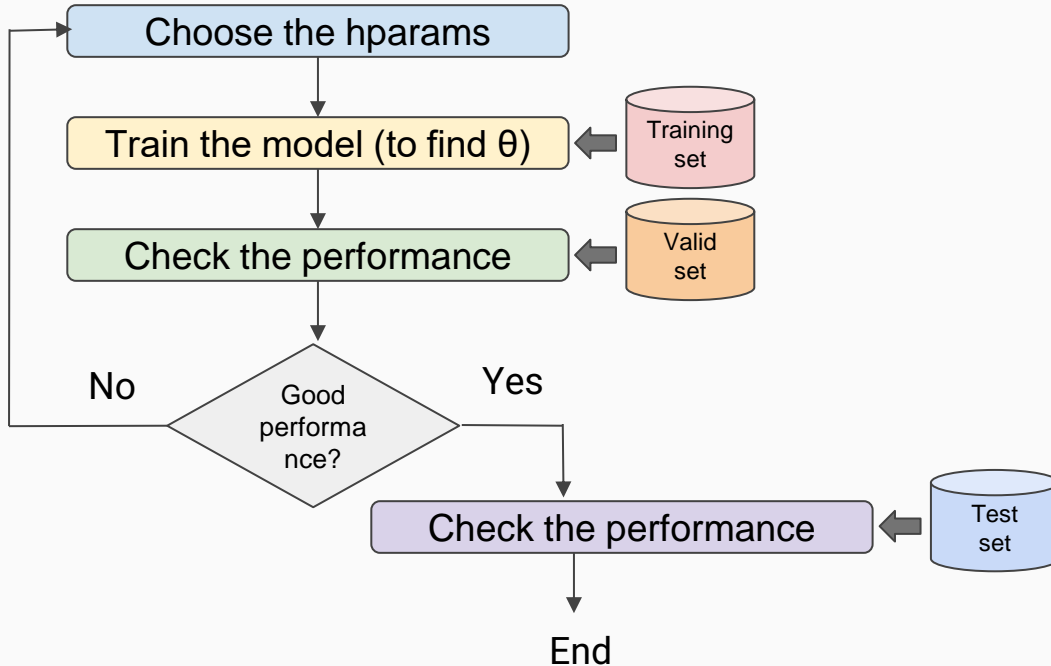
- Searching for the best hyperparameter is usually **expensive** because we have to **train the model multiple times** before finding the best configuration.
- One possible way is to initialize the hyperparameters with “reasonable” values (based on our experience or default settings suggested in the literature).
- Then, we can fine-tune the initial configuration through a **hyperparameter search**:

1. *Manual Search*
2. *Grid Search*
3. *Random Search*
4. *Bayesian Optimization*



Hyperparameter Search

- To summarize, this is what we do when we develop a machine learning model:



Hyperparameters

Parameters

- They are part of the model $f(\mathbf{x}, \theta)$.
- They are estimated during **training** using a **training set**.
- In gradient descent, we train the model by computing a gradient of the objective over the parameters.
- Examples of parameters are the weights w_0 and w_1 .

Hyperparameters

- They are external to the model $f(\mathbf{x}, \theta)$.
- They are not estimated during training, but during the **hyperparameter search** (performed on the **validation set**)
- We cannot compute the gradient over the hyperparameters.
- Examples are the learning rate, batch size, and number of epochs.

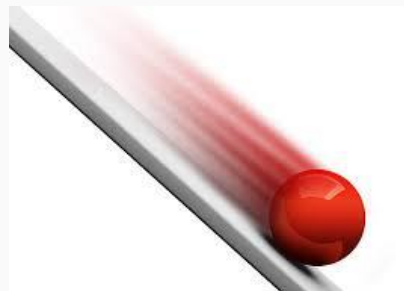
Basic Machine Learning Concepts:

Variants and Extensions of Gradient Descent

SGD Extensions and Variants

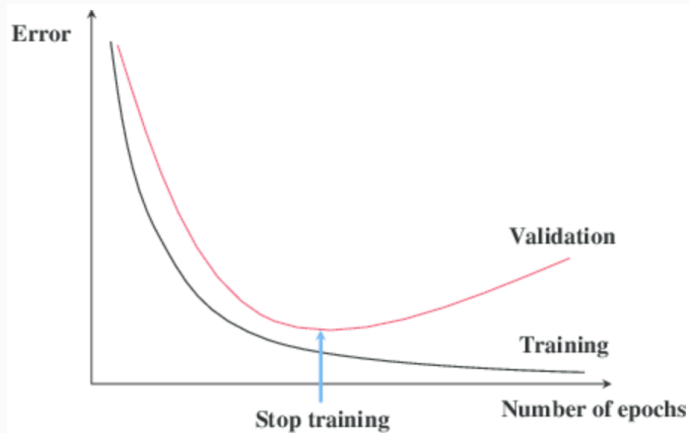
Several improvements have been proposed to the vanilla SGD algorithm:

- Early Stopping
- Learning Rate Annealing
- SGD with Momentum
- Adaptive Learning Rate methods (e.g, AdaGrad, RMSPROP, Adam)
- Second Order Methods (e.g., Newton's methods)



Early Stopping

- Often we iterate gradient descent for a predefined **number of epochs** (which is one of the hyperparameters of the system)
- However, if the number of epochs is *too high* we might end up in an **overfitting** regime.
- If the number of epochs is *too low* we might end up in an **underfitting** regime.

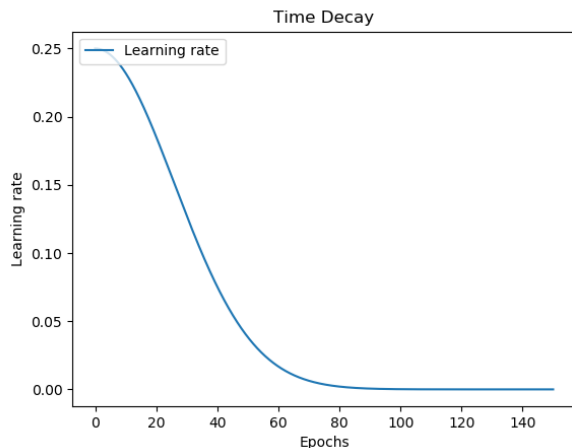


We can monitor the performance after each epoch on the validation set and stop training if the validation performance starts to get worse.

- This strategy is known as **early stopping**.
- It is one of the most commonly used forms of **regularization**.

Learning Rate Annealing

- Changing the learning rate over the epochs can improve performance and reduce training time.
- We normally **reduce the learning rate** while we train.
- This operation is known as **learning rate annealing**.

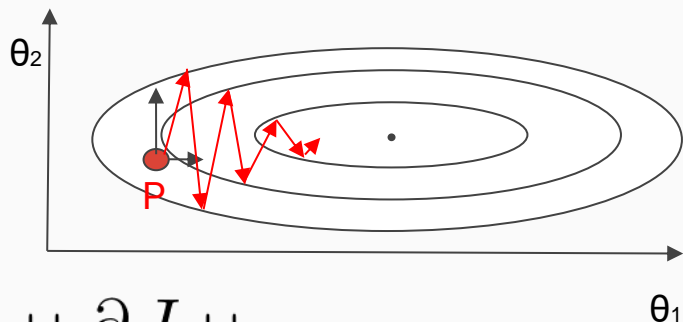


- Several methods have been proposed, such as *exponential decay*, *linear decay*, etc.
- Sometimes we reduce the learning rate when some conditions are met (e.g little improvement on the validation set). This is called *new-bob* annealing.

SGD with Momentum

Momentum

- SGD has trouble navigating in areas where the surface curves **much more steeply** in one dimension than in another.



$$\left\| \frac{\partial J}{\partial \theta_1} \right\| \quad \text{Small}$$

$$\left\| \frac{\partial J}{\partial \theta_2} \right\| \quad \text{Large}$$

- The partial derivative in the point P wrt θ_1 have a **small magnitude** because the area is rather flat in this dimension
- The partial derivative in the point P wrt θ_2 have a **large magnitude** because the area is quite steep in this dimension
- As a result, gradient descent will do a **little update** for θ_1 and a **larger** one for θ_2
- This causes a **slow convergence** as we lose time jumping back and forth for the sides of the narrow valley.

Momentum

- The method of momentum tackles this problem by accumulating an exponentially-decaying **moving average** of the **past gradients**:

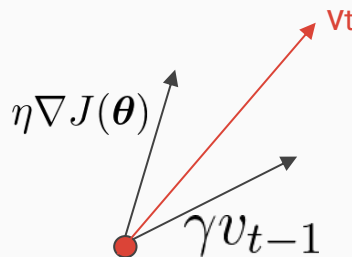
$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla J(\boldsymbol{\theta})$$

Diagram illustrating the components of the momentum update equation:

- \mathbf{v}_t : velocity
- $\gamma \mathbf{v}_{t-1}$: Momentum term (Previous velocity)
- η : Learning rate
- $\nabla J(\boldsymbol{\theta})$: Gradient

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \mathbf{v}_t$$

- The velocity \mathbf{v}_t is the vector containing the updates to perform.
- This time the updates do not only depend on the learning rate and the gradient, but also on the **previous updates** \mathbf{v}_{t-1} (weighted by a factor γ)



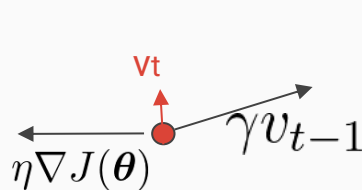
Momentum

- The method of momentum tackles this problem by accumulating an exponentially-decaying **moving average** of the **past gradients**:

velocity Momentum term Previous velocity Learning rate Gradient

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla J(\boldsymbol{\theta})$$

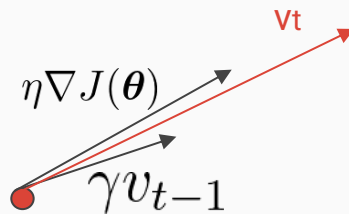
$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \mathbf{v}_t$$



If the previous update \mathbf{v}_t points in a direction very different from the current gradient.



I do a **little** update

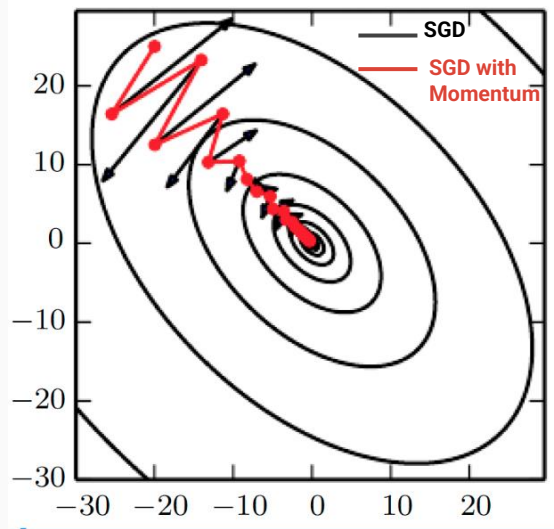


If the previous update \mathbf{v}_t points in a direction similar to the current gradient.



I do a **big** update

Momentum



- The effect of the momentum is to **dampen the oscillations** observed with SGD and reach the minimum faster.
- *Why?* Because two consecutive updates point in a very **different direction**.
- As we have seen, this leads to a **smaller update** that **minimizes the jumps** over the side of the valley.

The basic idea of the momentum is the following:

- When there is an “*agreement*” between the current and previous gradients, we are “*safe*” to do a big step.
- If there is “*disagreement*”, it is better to be “*prudent*” and do little steps.

Nesterov Momentum

- A popular variation of the standard momentum is the so-called **Nesterov Momentum**:

Standard Momentum

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \mathbf{v}_t$$

Nesterov Momentum

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla J(\boldsymbol{\theta} - \gamma \mathbf{v}_{t-1})$$

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \mathbf{v}_t$$

- We here try to “anticipate” the next move and apply a “correction factor” to the standard momentum.
- This anticipatory update prevents us from going too fast.

Adaptive Learning Rate

Adaptive Learning Rate

- In SGD, the **learning rate** η is the **same** for all the parameters $\theta = [\theta_1, \dots, \theta_i, \dots, \theta_P]^T$
- However, each parameter is different from the others.



Idea: *why not using a different learning rate for each parameter?*

Standard SGD

$$\theta_{new} = \theta - \eta \nabla J(\theta)$$

Constant Value

SGD with adaptive learning rate


$$\theta_{new} = \theta - \eta \odot \nabla J(\theta)$$

Vector

Element-wise multiplication

$$\eta = [\eta_1, \eta_2, \dots, \eta_P]^T$$

AdaGrad

- In a real case, we have **millions** or even **billions** of parameters  We cannot set their learning rate manually
- We need a method that assigns the learning rate to each parameter **automatically**.
- Adagrad (*Duchi et al., 2011*) proposed to do it in this way:

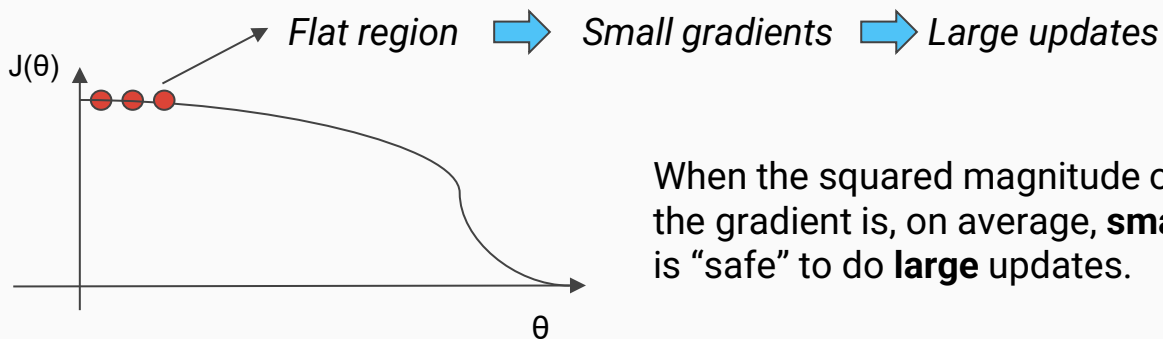
$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \frac{\eta}{\epsilon + \sqrt{\mathbf{r}_{new}}} \odot \nabla J(\boldsymbol{\theta}) \quad \mathbf{r}_{new} = \mathbf{r} + \nabla J(\boldsymbol{\theta}) \odot \nabla J(\boldsymbol{\theta})$$



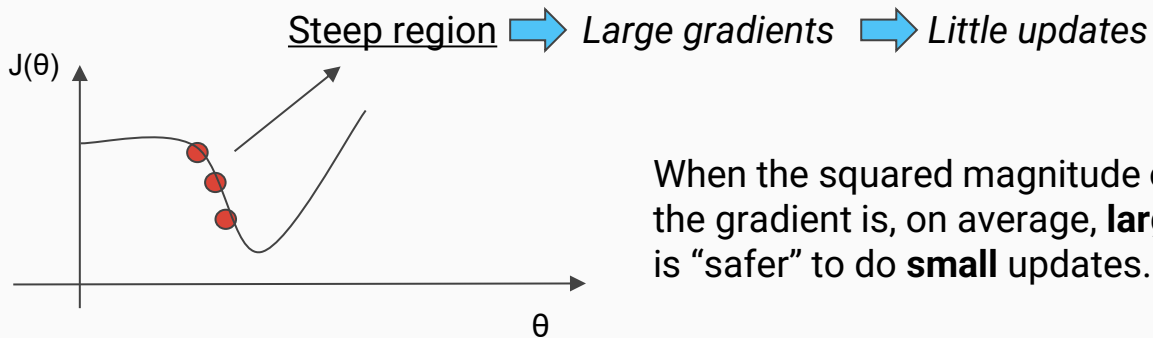
Small constant (for numerical stability)

We individually scale each parameter update using the **historical values** of the squared gradient magnitude.

AdaGrad



When the squared magnitude of the gradient is, on average, **small** it is "safe" to do **large** updates.



When the squared magnitude of the gradient is, on average, **large** it is "safer" to do **small** updates.

Adagrad

$$\theta_{new} = \theta - \frac{\eta}{\epsilon + \sqrt{\mathbf{r}_{new}}} \odot \nabla J(\theta)$$

$$\mathbf{r}_{new} = \mathbf{r} + \nabla J(\theta) \odot \nabla J(\theta)$$

Problem:

The updates tend to get smaller and smaller over time as **we keep accumulating the squared magnitude of gradients** (positive quantity)

This decrease is excessive for many practical applications

RMSProp

- **RMSProp** mitigates this issue by changing the squared gradient accumulation into an **exponential moving average**.

$$\begin{aligned}\mathbf{r}_{new} &= \overset{\text{Decay Rate}}{\rho} \mathbf{r} + (1 - \rho) \nabla J(\boldsymbol{\theta}) \odot \nabla J(\boldsymbol{\theta}) \\ \boldsymbol{\theta}_{new} &= \boldsymbol{\theta} - \frac{\eta}{\epsilon + \sqrt{\mathbf{r}_{new}}} \odot \nabla J(\boldsymbol{\theta})\end{aligned}$$

- With the exponential moving average, we give more “weight” to the most recent updates and less weight to the older ones.
- Good default values are $\eta = 0.001$ and $\rho = 0.9$

- RMSProp has been shown to **work well in practice** for real machine learning methods.

Adam

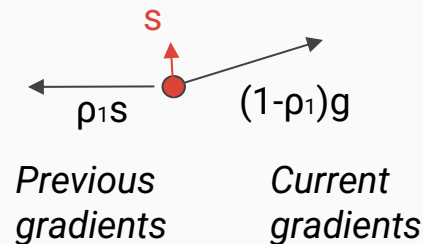
- **Adam** is an extension of the RMSProp optimizer that also considers momentum.

$$\mathbf{s}_{new} = \rho_1 \mathbf{s} + (1 - \rho_1) \nabla J(\boldsymbol{\theta})$$

$$\mathbf{r}_{new} = \rho_2 \mathbf{r} + (1 - \rho_2) \nabla J(\boldsymbol{\theta}) \odot \nabla J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \eta \frac{\mathbf{s}_{new}}{\sqrt{\mathbf{r}_{new} + \epsilon}}$$

Similar to **momentum** (but with exponential weighting)



The term s is big if the gradient points in the same directions, small if they point in different directions.

Adam

- **Adam** is an extension of the RMSProp optimizer that also considers momentum.

$$\mathbf{s}_{new} = \rho_1 \mathbf{s} + (1 - \rho_1) \nabla J(\boldsymbol{\theta})$$

$$\mathbf{r}_{new} = \rho_2 \mathbf{r} + (1 - \rho_2) \nabla J(\boldsymbol{\theta}) \odot \nabla J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \eta \frac{\mathbf{s}_{new}}{\sqrt{\mathbf{r}_{new} + \epsilon}}$$



This term is the same as the one used in RMSProp

The term \mathbf{r} is big if the squared magnitude of the gradient is big, and small otherwise.

Adam

- **Adam** is an extension of the RMSProp optimizer that also consider momentum.

$$\mathbf{s}_{new} = \rho_1 \mathbf{s} + (1 - \rho_1) \nabla J(\boldsymbol{\theta})$$

$$\mathbf{r}_{new} = \rho_2 \mathbf{r} + (1 - \rho_2) \nabla J(\boldsymbol{\theta}) \odot \nabla J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \eta \frac{\mathbf{s}_{new}}{\sqrt{\mathbf{r}_{new} + \epsilon}}$$

Suggested values:

$$\rho_1 = 0.9$$

$$\rho_2 = 0.999$$

$$\eta = 0.001$$



The update considers exponential moving averages of the gradients (first-order moment) and the magnitude of the gradients (second-order moment).

- With RMSProp, the direction of the update depends only on the current gradient ($\rho_1=0$), while the step size also depends on the history of the squared gradient.
- With Adam, both the direction of the update and the step size depends on the past gradients.

Adam

- Normally, \mathbf{s} and \mathbf{r} are initialized to 0.

$$\mathbf{s}_{new} = \rho_1 \mathbf{s} + (1 - \rho_1) \nabla J(\boldsymbol{\theta})$$

$$\mathbf{r}_{new} = \rho_2 \mathbf{r} + (1 - \rho_2) \nabla J(\boldsymbol{\theta}) \odot \nabla J(\boldsymbol{\theta})$$

This adds a **bias**, especially during the initial time steps, and when ρ_1 and ρ_2 are close to 1.

The bias correction has an effect only in the **first part of training**.

When t grows, \mathbf{s} and $\hat{\mathbf{s}}$ get closer and closer.

Correcting this bias is not that crucial in practice.

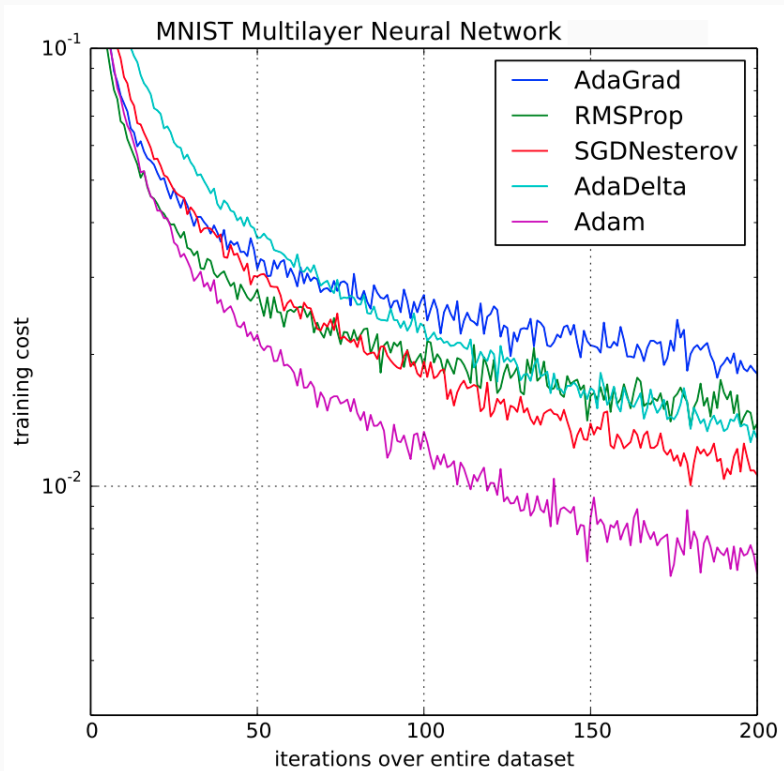
We can compensate for this bias in this way:

$$\hat{\mathbf{s}} = \frac{\mathbf{s}}{1 - \rho_1^t} \quad \hat{\mathbf{r}} = \frac{\mathbf{r}}{1 - \rho_2^t}$$

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \eta \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \epsilon}}$$

Where t is the update number (e.g., first update $t=1$, second $t=2$, etc)

Adam



Adam often works better than other optimizers in real machine learning problems.



It requires setting ρ_1 , ρ_2 , η . However, the default suggested values often work well ($\rho_1=0.9$, $\rho_2=0.999$, $\eta = 0.001$).



It requires storing s and r which are vectors of size corresponding to the number of parameters to optimize θ .

If we have millions or billions of parameters, this can be quite memory-demanding.

Second Order Methods

Second Order Methods

- We have seen that the gradient (based on first order partial derivatives) provides useful information that we can use to minimize our objective.



What about using the **second order derivative**?

Gradient

$$\nabla J(\boldsymbol{\theta}) = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \dots, \frac{\partial J}{\partial \theta_M} \right]^T$$

$$\nabla J(\boldsymbol{\theta}) \in \mathbb{R}^P$$

The gradient is a vector containing the **first order partial derivatives**.

Hessian

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 J}{\partial \theta_1^2} & \frac{\partial^2 J}{\partial \theta_1 \theta_2} & \cdots & \frac{\partial^2 J}{\partial \theta_1 \theta_M} \\ \frac{\partial^2 J}{\partial \theta_2 \theta_1} & \frac{\partial^2 J}{\partial \theta_2^2} & \cdots & \frac{\partial^2 J}{\partial \theta_2 \theta_M} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 J}{\partial \theta_P \theta_1} & \frac{\partial^2 J}{\partial \theta_P \theta_2} & \cdots & \frac{\partial^2 J}{\partial \theta_P^2} \end{bmatrix}$$

The Hessian is a **symmetric** square matrix of **second-order partial derivatives**. $\mathbf{H} \in \mathbb{R}^{M \times M}$

Second Order Methods

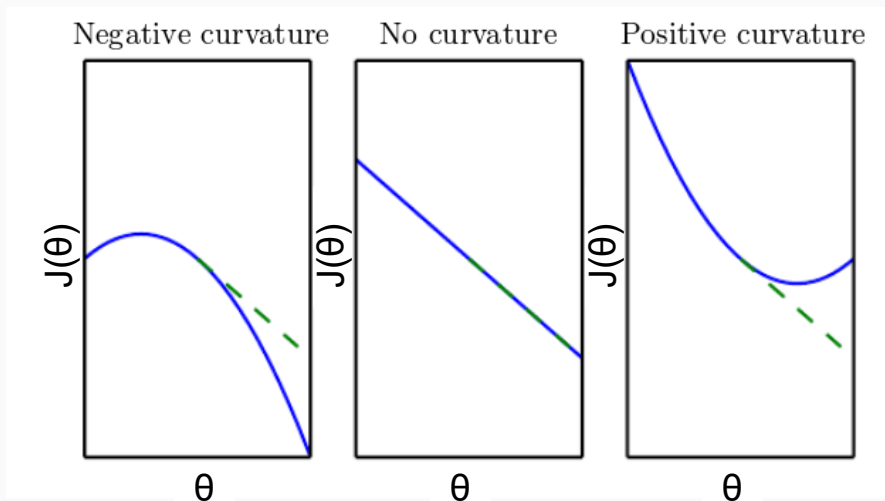
- The second order derivative tells us how the derivative changes when applying a little change in input x .

Hessian

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 J}{\partial \theta_1^2} & \frac{\partial^2 J}{\partial \theta_1 \theta_2} & \cdots & \frac{\partial^2 J}{\partial \theta_1 \theta_M} \\ \frac{\partial^2 J}{\partial \theta_2 \theta_1} & \frac{\partial^2 J}{\partial \theta_2^2} & \cdots & \frac{\partial^2 J}{\partial \theta_2 \theta_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 J}{\partial \theta_P \theta_1} & \frac{\partial^2 J}{\partial \theta_P \theta_2} & \cdots & \frac{\partial^2 J}{\partial \theta_P^2} \end{bmatrix}$$

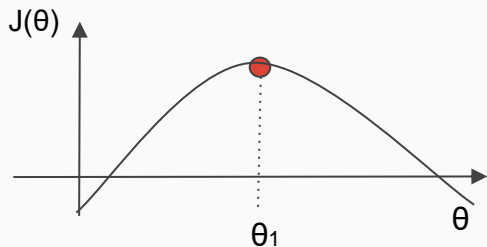
The Hessian is a square matrix of **second-order partial derivatives**.

- It measures the **curvature** of the objective function $J(\theta)$ around the point θ .

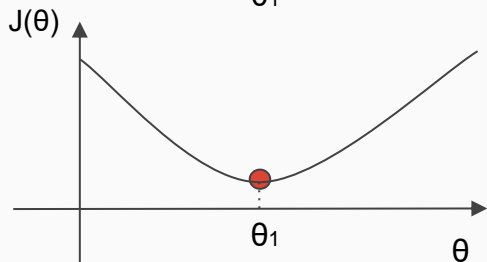


Second Order Methods

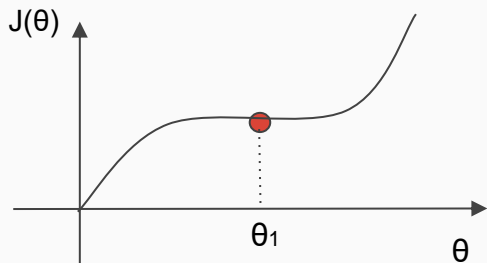
The information of the second derivative can be used to classify critical points (**second derivative test**)



$$\frac{\partial J(\theta_1)}{\partial \theta} = 0 \quad \frac{\partial^2 J(\theta_1)}{\partial \theta^2} < 0 \quad \Rightarrow \quad \text{Local Maximum}$$



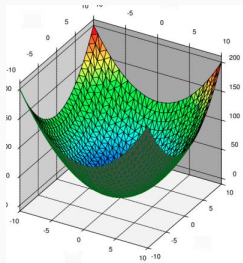
$$\frac{\partial J(\theta_1)}{\partial \theta} = 0 \quad \frac{\partial^2 J(\theta_1)}{\partial \theta^2} > 0 \quad \Rightarrow \quad \text{Local Minimum}$$



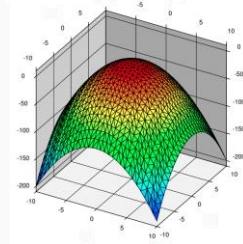
$$\frac{\partial J(\theta_1)}{\partial \theta} = 0 \quad \frac{\partial^2 J(\theta_1)}{\partial \theta^2} = 0 \quad \Rightarrow \quad \text{Inconclusive Saddle Point or flat region?}$$

Second Order Methods

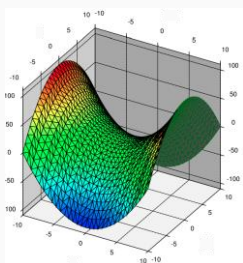
In a multi-dimensional case, the test involves the **eigenvalues**



$$H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \lambda = [2, 2] \quad \text{If **all** eigenvalues are **positive**} \Rightarrow \text{Local Minimum}$$



$$H = \begin{bmatrix} -2 & 0 \\ 0 & -2 \end{bmatrix} \quad \lambda = [-2, -2] \quad \text{If **all** eigenvalues are **negative**} \Rightarrow \text{Local Maximum}$$

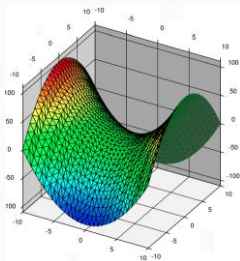


$$H = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix} \quad \lambda = [2, -2] \quad \text{If **at least one** eigenvalue is **positive** and at least one is **negative**} \Rightarrow \text{Saddle Point}$$

In this **special** case, we have a diagonal matrix and the eigenvalues are just the elements on the diagonal.

Second Order Methods

- The condition to have a saddle point is **less restrictive** than that needed for local minima and maxima.
- Intuitively, it will be significantly easier to find points with at least one eigenvalues is positive and at least one negative (**saddle points**) rather than finding points with all eigenvalues are positive (**minima**) or negative (**maxima**).
- Now, we can understand better why **saddle points** are much more **common** than local minima and maxima in **high-dimensional spaces**.

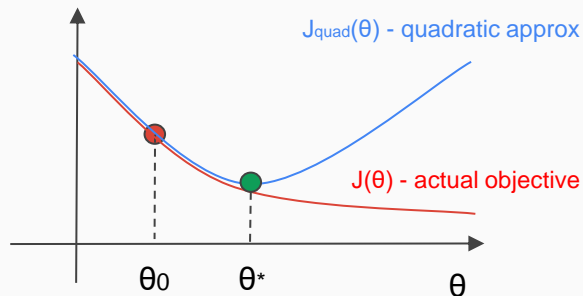


Newton's Method

- The optimization methods that use both the gradient and the Hessian are called **second order methods**.
- A popular one is called **Newton's method**.



We can approximate the objective with a **Taylor expansion** (up to the *second order*) and jump directly into the expected minimum value.



$$J(\theta) \approx J(\theta_0) + J'(\theta)(\theta - \theta_0) + \frac{1}{2}J''(\theta)(\theta - \theta_0)^2$$

Quadratic function

If the function is **convex** around θ_0 (*positive second derivative*), we can find the minimum by solving:

$$J'(\theta) = 0$$

Newton's Method

$$J(\theta) \approx J(\theta_0) + J'(\theta)(\theta - \theta_0) + \frac{1}{2}J''(\theta)(\theta - \theta_0)^2$$

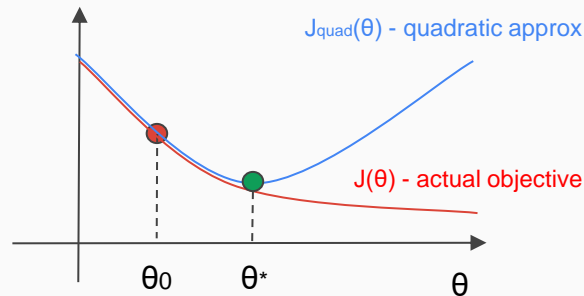
Quadratic function

Minimum

$$J'(\theta) = 0$$

$$J'(\theta_0) + J''(\theta_0)(\theta - \theta_0) = 0$$

$$\theta^* = \theta_0 - \frac{J'(\theta_0)}{J''(\theta_0)}$$



- The update equation is similar to gradient descent.
- The main difference is that the “learning rate” is not specified but automatically guessed using by the second derivative.
- Similar to gradient descent, we can iterate multiple times until convergence

Newton's Method

High-dimensional case

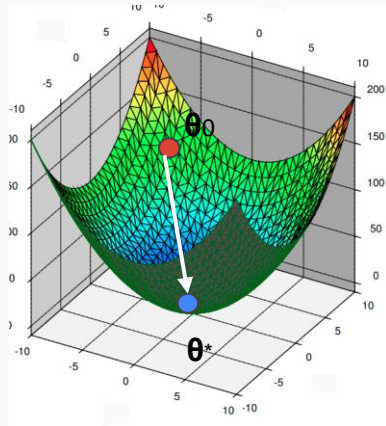
Quadratic approximation

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

$$\nabla J(\boldsymbol{\theta}) = 0 \quad \nabla J(\boldsymbol{\theta}_0) + \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

If \mathbf{H} is **positive definite**, this will find the **minimum**.

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla J(\boldsymbol{\theta}_0)$$



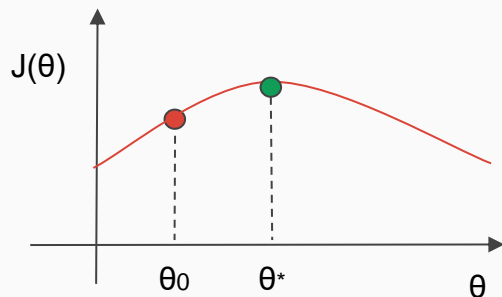
Algorithm:

1. Compute the Gradient.
2. Compute the Hessian
3. Compute Hessian Inverse.
4. Update parameters.

Newton's Method

Issues

- Newton's method sounds appealing, but in practice, it suffers from **several issues**:



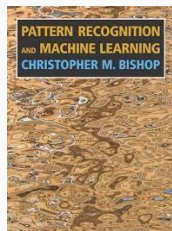
- If the second derivative is negative, we reach a **local maximum** and not a local minimum.
- In a multi-dimensional case, this happens if the eigenvalues of H are **not all positives**.
- This happens for instance, near **saddle points**.
- Newton's method is thus **sensitive to saddle points** (that are a big issue in high-dimensional spaces).
- Standard SGD is less sensitive to this issue.
- Several solutions have been proposed to mitigate this issue (e.g, Hessian regularization).

Newton's Method

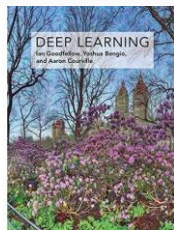
Issues

- The other issue is the **computational complexity** that increases significantly with the number of parameters:
 - Gradient Computation: $O(N)$
 - Hessian Computation: $O(N^2)$
 - Hessian Inversion: $O(N^3)$
- **Quasi-Newton methods** attempt to the computational burden by approximating the Hessian.
- Examples of techniques are *Conjugate Gradients* and the *BFGS Algorithm*.
- The objective should be twice differentiable.
- Numerical instability with second derivative close to zero.

Additional Material



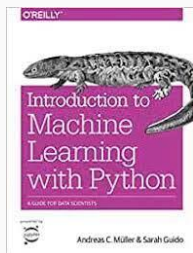
Introduction (page 1-11)



Chapter 2: Linear Algebra

Chapter 3: Probability and
information theory

Chapter 5: Machine Learning Basics



Introduction (page 1-27)

Linear Models (page 47-57)

Lab Session

- During the weekly lab session, we will do:

The logo for Google Colab, featuring the word "colab" in a stylized orange font.

Tutorial on Matplotlib

The logo for Google Colab, featuring the word "colab" in a stylized orange font.

Tutorial on Scikit-learn



Plotting exercises

The logo for Matplotlib, featuring the word "matplotlib" in a blue sans-serif font, with a circular icon containing a multi-colored star or flower-like shape.