

**Vietnam General Confederation of Labor
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY**



FINAL REPORT

MACHINE LEARNING

Instructor: **PhD LE ANH CUONG**

Student: **Do Minh Quan – 521H0290**

Class : **21H50301**

Year : **25**

HO CHI MINH CITY, 2023

Vietnam General Confederation of Labor
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY



FINAL REPORT

MACHINE LEARNING

Instructor: **PhD LE ANH CUONG**

Student: **Do Minh Quan – 521H0290**

Class : **21H50301**

Year : **25**

HO CHI MINH CITY, 2023

ACKNOWLEDGEMENT

I express my sincere appreciation to Dr. Le Anh Cuong from Ton Duc Thang University for his invaluable guidance during the development of this web application. His expertise in optimizing machine learning models has significantly influenced the project's direction.

Dr. Le Anh Cuong's commitment to excellence and his willingness to share his knowledge have played a crucial role in the success of this endeavor. His mentorship has deepened my understanding of various optimizer methods in machine learning, contributing to the project's overall quality.

I extend my heartfelt thanks to Dr. Le Anh Cuong for his continuous encouragement and support, pivotal in completing this project. His dedication to creating a conducive learning environment has been a great inspiration, and I am grateful for the opportunity to work under his guidance.

This acknowledgement reflects the influence of Dr. Le Anh Cuong's expertise not only in the realm of machine learning optimization but also in fostering continual learning and test production strategies in real-world problem-solving scenarios.

Ho Chi Minh city, 18th December, 2023

Author

(Sign and write full name)



Do Minh Quan

THIS PROJECT WAS COMPLETED AT TON DUC THANG UNIVERSITY

I fully declare that this is my own project and is guided by PhD Le Anh Cuong; The research contents and results in this topic are honest and have not been published in any form before. The data in the tables for analysis, comments and evaluation are collected by the author himself from different sources, clearly stated in the reference section.

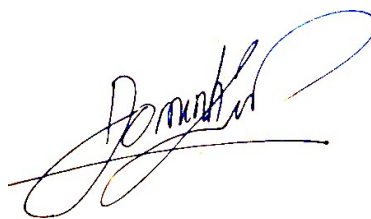
Besides that, the project also uses a number of comments, assessments as well as data from other authors, other agencies and organizations, with citations and source annotations.

Should any frauds were found, I will take full responsibility for the content of my report. Ton Duc Thang University is not related to copyright and copyright violations caused by me during the implementation process (if any).

Ho Chi Minh city, 18th December, 2023

Author

(Sign and write full name)



Do Minh Quan

CONFIRMATION AND ASSESSMENT SECTION

Instructor confirmation section

Ho Chi Minh December, 2023
(Sign and write full name)

Evaluation section for grading instructor

Ho Chi Minh December, 2023
(Sign and write full name)

SUMMARY

This report delves into a comprehensive exploration and comparative analysis of various optimizer methods employed in the training of machine learning models. The study investigates the efficacy of different optimization techniques, aiming to enhance our understanding of their impact on model performance and convergence. By scrutinizing these methods, the report aims to provide insights into the strengths, weaknesses, and applicability of each optimizer, thus aiding practitioners in making informed decisions when selecting optimization strategies for their machine learning projects.

Furthermore, the report delves into the realm of Continual Learning, shedding light on its significance and implications in the context of building machine learning solutions. Continual Learning, a dynamic approach to model training, is explored for its potential to adapt to evolving data distributions and sustain performance over time. The report examines key principles, challenges, and emerging trends in Continual Learning, offering a holistic view of its integration into the machine learning pipeline.

Additionally, the report investigates Test Production in the context of constructing a machine learning solution to address specific problem domains. Test Production, a critical aspect of model evaluation and validation, is explored for its role in ensuring the robustness and generalization of machine learning models. The study explores methodologies and best practices associated with Test Production, aiming to equip practitioners with the knowledge needed to create reliable and effective machine learning solutions.

In conclusion, this report synthesizes valuable insights into optimizer methods, Continual Learning, and Test Production, providing a comprehensive overview for researchers, practitioners, and enthusiasts in the field of machine learning. The findings contribute to the ongoing discourse on optimization strategies and model development, fostering a deeper understanding of these crucial components in the pursuit of more robust and adaptable machine learning solutions.

1.2.5.6 Adamax	19
1.3 Comparative Analysis	20
CHAPTER 2 – CONTINUAL LEARNING AND TEST PRODUCTION IN MACHINE LEARNING SOLUTIONS	22
2.1 Continual Learning.	22
2.1.1 Introduction.....	22
2.1.2 The Continuous Learning Process	24
2.1.3 Types of continuous machine learning	25
2.1.4 How Continuous learning process work in machine learning	27
2.1.5 Advantages and limitations of continuous learning process.....	28
2.1.6 Applications of Continuous Learning.....	30
2.2 Test Production	30
2.2.1 Introduction	30
2.2.2 The testing process.....	31
2.2.2.1 Model Evaluation.....	31
2.2.2.2 Pre-Training Test/ Unit Testing.....	31
2.2.2.3 Post Training Test- Batch Vs Online Learning	32
2.2.2.4 A/B testing	33
2.2.2.5 Stage test/ Shadow test	34
2.2.2.6 API Testing	34
2.2.3 Types of testing production	34
CHAPTER 3 – REFERENCES	36

LIST OF ABBREVIATIONS

- 1 SGD: Stochastic Gradient Descent
- 2 GC: Gradient Centralization
- 3 Adagrad: Adaptive Gradient Algorithm
- 4 Adadelat: Adaptive Delta
- 5 RMSProp: Root Mean Square Propagation
- 6 Adam: Adaptive Moment Estimation
- 7 Adamax: Adaptive Moment Estimation with infinity norm

LIST OF FIGURES

Figure 1: Attaining Global minimum before and after scaling	7
Figure 2: Before and after using Standardization.....	7
Figure 3: Gradient Descent.....	8
Figure 4: Types of Gradient Descent.....	9
Figure 5: SGD with momentum and with momentum	11
Figure 6: Comparing Performances of Different Adaptive Learning Algorithms	14
Figure 7: Definitions – CI, CT, CD, and CML	23
Figure 8: Flowchart of the Continuous Learning Process to Machine Learning	25
Figure 9: Machine learning testing flow.....	31
Figure 10: Unit testing.....	32
Figure 11: Post training test.....	33
Figure 12: A/B testing	33
Figure 13: Stage test	34
Figure 14: API Testing	34
Figure 15: Type of tests	35

LIST OF TABLES

Table 1 Pros and cons Optimizers	22
--	----

CHAPTER 1 – COMPARATIVE ANALYSIS OF OPTIMIZER METHODS IN MACHINE LEARNING MODEL TRAINING

1.1 Introduction Optimizers

Machine learning optimization is a crucial process aimed at improving the accuracy of a model by minimizing the error between predicted and true values. The models learn to generalize from training data, approximating the underlying relationship between input and output. The optimization process involves iteratively adjusting parameters to minimize a loss or cost function, which measures the disparity between predicted and actual values.

Data preparation is considered an optimization step, transforming raw, unlabelled data into training data usable by the model. However, the primary focus of machine learning optimization is on hyperparameter tuning. Hyperparameters, set by data scientists, are not learned from data but are crucial for configuring the model effectively. They include learning rates, model structure, and cluster counts, shaping the model for specific tasks.

Hyperparameter optimization ensures the model efficiently and accurately solves the intended problem. It involves tweaking configurations to align with specific goals, promoting efficiency and effectiveness. Achieving the most accurate model involves tuning hyperparameters through a process known as hyperparameter tuning or optimization, aiming for maximum accuracy and efficiency while minimizing errors.

1.2 Popular Optimizer Methods

1.2.1 Feature Scaling

Feature scaling is a process that is performed prior to data processing on independent variables or characteristics. This is done because many times the data set will contain highly variable characteristics in magnitudes, units or ranges. Feature scaling basically helps to normalize the data within a particular range.

Techniques to perform Feature Scaling. Consider the two most important ones:

- **Min-Max Normalization:** This technique re-scales a feature or observation value with distribution value between 0 and 1.

$$X_{new} = \frac{X_i - \min(X)}{\max(X) - \min(X)}$$

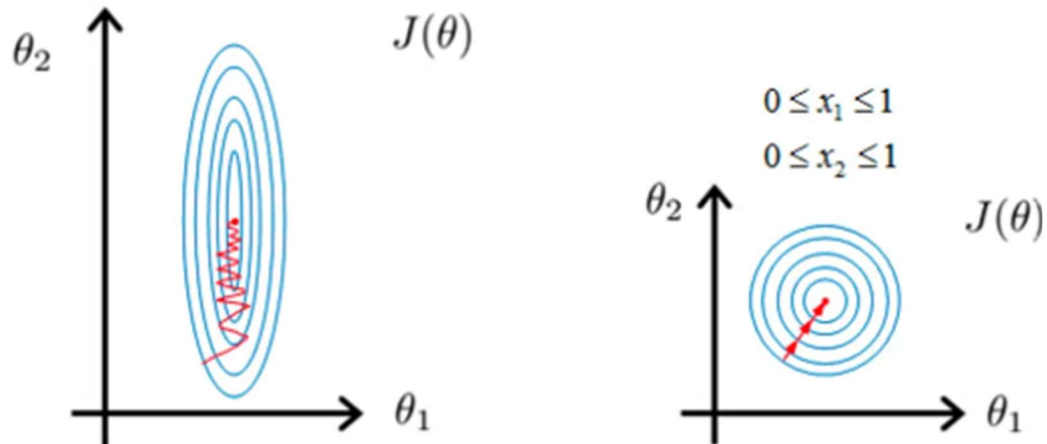


Figure 1: Attaining Global minimum before and after scaling

- **Standardization:** It is a very effective technique which re-scales a feature value so that it has distribution with 0 mean value and variance equals to 1.

$$X_{new} = \frac{X_i - X_{mean}}{\text{standard Deviation}}$$

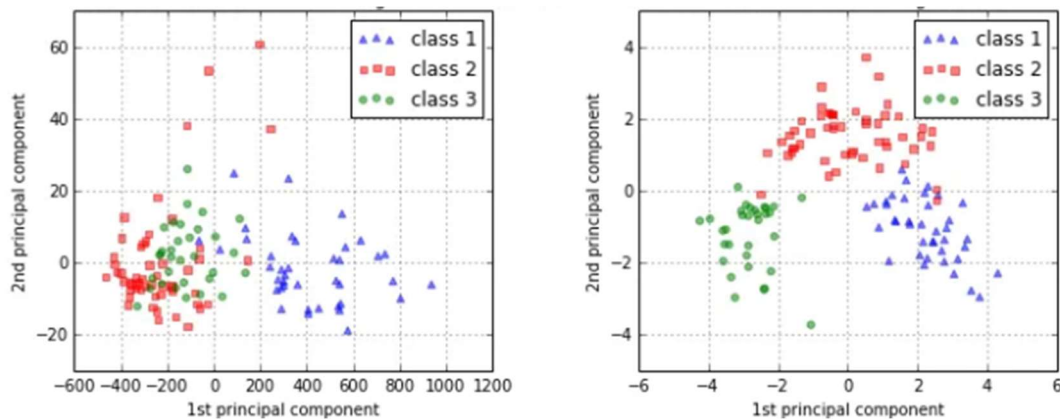


Figure 2: Before and after using Standardization

1.2.2 Gradient Descent

1.2.2.1 Introduction

The gradient descent algorithm is an optimization technique largely utilized in machine learning and deep learning. Gradient descent modifies parameters to reduce certain functions to local minima. In linear regression, it discovers weight and biases, and deep learning backward propagation employs the procedure. The method purpose is to discover model parameters like weight and bias that decrease model error on training data.

How its work :

Instead of trekking up a hill, think of gradient drop as hiking down to the bottom of a valley. This is a better example because it is a minimization method that minimizes a specified function. The

equation below illustrates what the gradient descent method does: b is the future location of our climber, while a represents his current position. The negative sign refers to the minimization stage of the gradient descent technique. The gamma in the center is a waiting factor and the gradient term ($\nabla f(a)$) is just the direction of the sharpest decline.

$$b = a - \gamma \nabla f(a)$$

So this algorithm effectively tells us the next location we need to travel, which is the direction of the sharpest slope. Let's look at another example to truly hammer the notion home.

Imagine you have a machine learning issue and wish to train your algorithm with gradient descent to minimize your cost-function $J(w, b)$ and attain its local minimum by modifying its parameters (w and b). The picture below displays the horizontal axes showing the parameters (w and b), while the cost function $J(w, b)$ is depicted on the vertical axis. Gradient descent is a convex function.

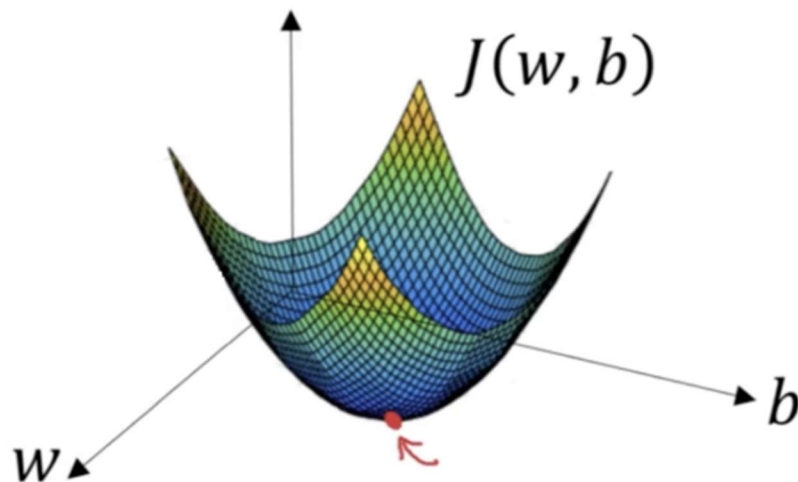


Figure 3: Gradient Descent

We know we want to discover the values of w and b that correspond to the minimum of the cost function (shown with the red arrow). To start finding the proper values we initialize w and b with some random integers. Gradient descent then starts at that point (somewhere around the top of our picture), and it continues one step after another in the steepest downhill direction (i.e., from the top to the bottom of the illustration) until it reaches the point where the cost function is as minimal as feasible.

Types of Gradient Descent:

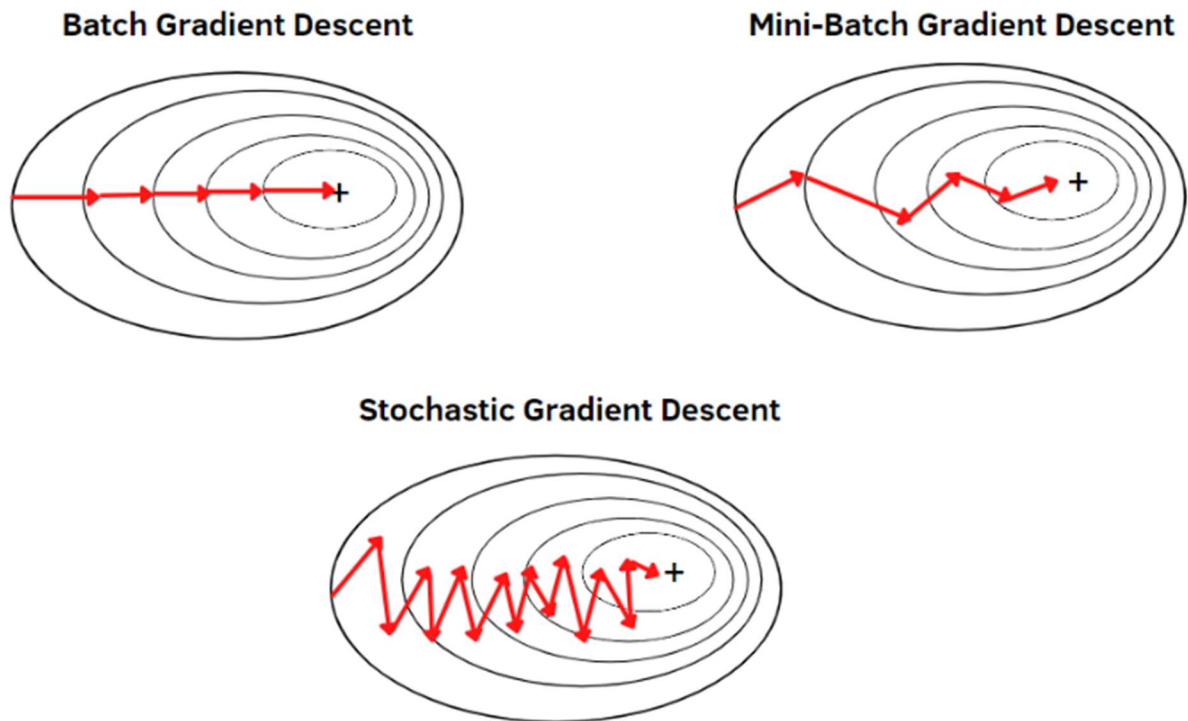


Figure 4: Types of Gradient Descent

1.2.2.2 Batch Gradient Descent

Batch gradient descent, also termed vanilla gradient descent, evaluates the error for each example inside the training dataset, but only after all training examples have been reviewed does the model get changed. This whole procedure is like a circle and it's termed a training period.

Some advantages of batch gradient descent include its computing efficiency: it creates a steady error gradient and a stable convergence. Some downsides include that the stable error gradient might occasionally result in a state of convergence that isn't the best the model can accomplish. It also requires the complete training dataset to be in memory and available to the algorithm.

1.2.2.3 Mini- Batch Gradient Descent

Mini-batch gradient descent is the go-to method since it's a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

Common mini-batch sizes range between 50 and 256, but like any other machine learning technique, there is no clear rule because it varies for different applications. This is the go-to algorithm when training a neural network and it is the most common type of gradient descent within

deep learning.

1.2.2.4 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is an iterative optimization technique extensively employed in machine learning and deep learning. It is a version of gradient descent that provides updates to the model parameters (weights) based on the gradient of the loss function computed on a randomly selected portion of the training data, rather than on the complete dataset.

The core principle of SGD is to pick a tiny random portion of the training data, termed a mini-batch, and compute the gradient of the loss function with respect to the model parameters using only that fraction. This gradient is then utilized to update the parameters. The procedure is continued with a new random mini-batch until the algorithm converges or achieves a predetermined stopping condition.

SGD provides various advantages over traditional gradient descent, such as quicker convergence and lower memory needs, especially for big datasets. It is also more resilient to noisy and non-stationary data, and can escape from local minima. However, it may need more iterations to converge than gradient descent, and the learning rate needs to be carefully calibrated to assure convergence.

Stochastic Gradient Descent is an iterative optimization technique that uses minibatches of data to form an expectation of the gradient, rather than the full gradient using all available data. That is for weights (W) and a loss function L we have:

$$W_{t+1} = W_t - n \nabla_w L_{W(t)}$$

Where n is a learning rate. SGD reduces redundancy compared to batch gradient descent - which recomputes gradients for similar examples before each parameter update - so it is usually much faster.

1.2.1.4 Stochastic Gradient Descent with Gradient Clipping (SGD with GC)

Stochastic Gradient Descent with gradient clipping (SGD with GC) is a version of the conventional SGD technique that includes an extra step to prevent the gradients from getting too big during training, which can cause instability and sluggish convergence.

Gradient clipping includes scaling down the gradients if their norm exceeds a predetermined threshold. This helps to prevent the "exploding gradient" problem, which can arise when the gradients get too big and force the weights to update too much in a single step.

In SGD with GC, the method computes the gradients on a randomly selected mini-batch of

training samples, as in regular SGD. However, before applying the gradients to update the model parameters, the gradients are trimmed if their norm exceeds a preset threshold. This threshold is often set to a small value, such as 1.0 or 5.0.

The gradient clipping step can be used either before or after any regularization procedures, such as L2 regularization. It is also usual to utilize adjustable learning rate algorithms, such as Adam, in combination with SGD with GC to further increase convergence.

SGD with GC is particularly effective when training deep neural networks, where the gradients can rapidly become unstable and create convergence issues. By restricting the magnitude of the gradients, the method can converge quicker and with more stability, resulting to increased performance on the test set.

1.2.4 Momentum-Based Optimizers.

1.2.4.1 Introduction

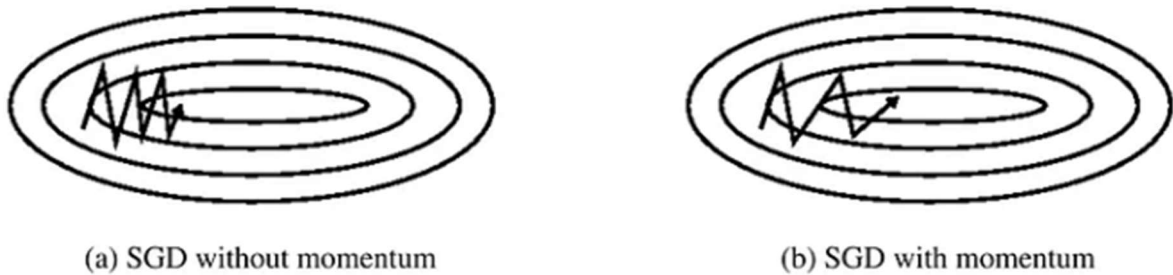


Figure 5: SGD with momentum and with momentum

Observation:

SGD without momentum , Updates takes longer vertical steps [slower learning]than horizontal step[faster learning].

SGD with momentum,Updates takes longer Horizontal steps[faster learning] then vertical step[slower learning].

Problem : Momentum based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley Takes a lot of u-turns before finally converging Despite these u-turns it still converges faster than vanilla gradient descent.

1.2.4.1 Momentum

Momentum is an optimization strategy used in machine learning and deep learning to expedite the training of neural networks. It is based on the notion of adding a portion of the previous update to the current update of the weights throughout the optimization process.

In momentum optimization, the gradient of the cost function is computed with respect to each

weight in the neural network. Instead of updating the weights directly based on the gradient, momentum optimization introduces a new variable, termed the momentum term, which is utilized to update the weights. The momentum term is a moving average of the gradients, and it accumulates the prior gradients to assist influence the search direction.

The momentum term might be viewed as the velocity of the optimizer. The optimizer acquires momentum as it goes downhill and serves to attenuate oscillations in the optimization process. This can enable the optimizer to converge faster and to reach a better local minimum.

Momentum optimization is especially beneficial in circumstances when the optimization landscape is noisy or where the gradients vary fast. It can also assist to smooth out the optimization process and avoid the optimizer from becoming trapped in local minima.

How its work

Momentum in optimization is like a ball rolling downhill. While gradient descent updates parameters based on the current gradient, momentum adds a fraction of the previous update to the current one. This helps the optimizer persist in the same direction, reducing oscillations and preventing getting stuck in shallow local minima, addressing drawbacks of traditional gradient descent. The update rule for momentum can be written as follows :

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

$$\theta = \theta - \alpha v_t$$

- $J(\theta)$ is the cost function.
- $\nabla_{\theta} J(\theta)$ is the gradient of the cost function with respect to the parameters θ
- β is the momentum term
- v is the velocity or momentum vector
- α is the learning rate

The variable v represents the momentum term, β is the momentum coefficient, $J()$ is the gradient of the cost function with respect to the parameters and α is the learning rate in this equation. Typically, the momentum coefficient is set at 0.9. The optimizer calculates the gradient of the cost function at each iteration and updates the momentum term as the exponentially weighted moving average of the previous gradients. The parameters are then updated by removing the momentum term multiplied by the learning rate.

Overall, momentum is a potent optimization strategy that may assist expedite the training of deep neural networks and increase their performance.

1.2.4.2 Nesterov Momentum

Nesterov momentum is a variation of the momentum optimization approach used in machine learning and deep learning to expedite the training of neural networks. It is named after the mathematician Yurii Nesterov, who initially presented the notion.

In traditional momentum optimization, the gradient of the cost function is computed with respect to each weight in the neural network, and the weights are updated depending on the gradient and the momentum term. Nesterov momentum optimization changes this by first updating the weights with a fraction of the prior momentum term and then computing the gradient of the cost function at the new position.

The theory behind Nesterov momentum is that the momentum term can help to forecast the future position of the weights, which can then be utilized to compute a more accurate gradient. This can assist the optimizer to take bigger steps in the right direction and converge faster than normal momentum optimization.

Nesterov momentum is particularly effective in circumstances where the optimization terrain is quite rough or when the gradients vary fast. It can also assist to avoid the optimizer from overshooting the ideal solution and thus lead to improved convergence.

In Nesterov Momentum, the update involves a "lookahead" step before calculating the gradient. The parameters are first nudged in the direction of the previous momentum term, and then the gradient is calculated at this "lookahead" point.

$$v = v \beta + (1 - \beta) \nabla_{\theta} J(\theta - \beta v_{t-1})$$

$$\theta = \theta - \alpha v$$

The fundamental distinction is that the gradient is determined at the "lookahead" location $\theta - \beta v_{t-1}$ instead of at the current position θ . This allows the algorithm to make a more informed update, considering the momentum's influence in the direction of the future parameter update.

Overall, Nesterov momentum is a potent optimization strategy that may assist expedite the training of deep neural networks and increase their performance, particularly in hard optimization settings.

1.2.5 Adaptive Learning Rate Optimizers:

1.2.5.1 Introduction

The issue of employing learning rate schedules is that their hyperparameters have to be established in advance and they vary substantially on the kind of model and application. Another difficulty is that the same learning rate is applied to all parameter changes. If we have sparse data, we may wish to update the parameters in various extent instead.

Adaptive gradient descent methods such as Adagrad, Adadelata, RMSprop, Adam, give an alternative to traditional SGD. These per-parameter learning rate approaches give heuristic approach without needing significant labor in tweaking hyperparameters for the learning rate schedule manually.

In Keras, we can implement these adaptive learning algorithms easily using corresponding optimizers. It is usually recommended to leave the hyperparameters of these optimizers at their default values.

```
keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)
keras.optimizers.Adadelata(lr=1.0, rho=0.95, epsilon=1e-08, decay=0.0)
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
decay=0.0)
```

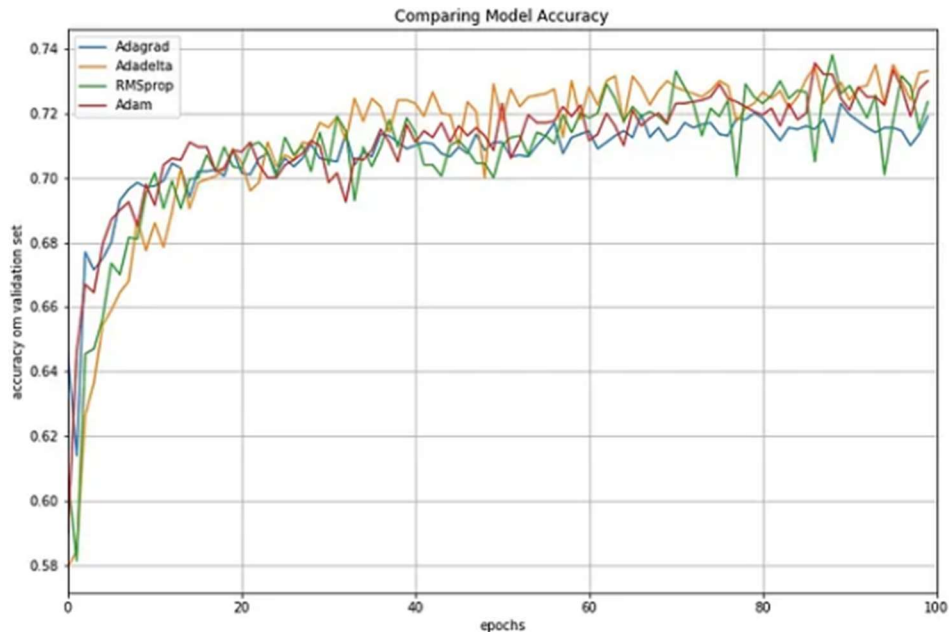


Figure 6: Comparing Performances of Different Adaptive Learning Algorithms

1.2.5.2 Adagrad (Adaptive Gradient)

Adagrad (Adaptive Gradient) is an optimization technique used in machine learning and deep learning to optimize the training of neural networks.

The Adagrad method modifies the learning rate of each parameter of the neural network adaptively during the training process. Specifically, it adjusts the learning rate of each parameter based on the previous gradients obtained for that parameter. In other words, parameters that have big gradients are given a lesser learning rate, whereas those with minor gradients are given a larger learning rate. This helps prevent the learning rate from lowering too quickly for frequently occurring parameters and allows for faster convergence of the training process.

The Adagrad technique is particularly effective for dealing with sparse data, when parts of the input characteristics have low frequency or are absent. In these instances, Adagrad is able to adaptively alter the learning rate of each parameter, which allows for better handling of the sparse data. Overall, Adagrad is a strong optimization method that may assist speed the training of deep neural networks and enhance their performance.

How its work : AdaGrad's main principle is to scale the learning rate for each parameter according to the total of squared gradients observed during training. The steps of the algorithm are as follows:

- Step 1: Initialize variables

Initialize the parameters θ and a small constant ϵ to avoid division by zero.

Initialize the sum of squared gradients variable G with zeros, which has the same shape as θ .

- Step 2: Calculate gradients

Compute the gradient of the loss function with respect to each parameter, $\nabla \theta J(\theta)$

- Step 3: Accumulate squared gradients

Update the sum of squared gradients G for each parameter i : $G[i] += (\nabla \theta J(\theta[i]))^2$

- Step 4: Update parameters:

Update each parameter using the adaptive learning rate: $\theta[i] -= (\eta / (\sqrt{G[i]} + \epsilon)) * \nabla \theta J(\theta[i])$

η :the learning rate

$\nabla \theta J(\theta[i])$: the gradient of the loss function with respect to parameter $\theta[i]$.

1.2.5.3 Adadelata

Adadelata is an optimization technique used in machine learning and deep learning to optimize the

training of neural networks. It is a variation of the Adagrad method that overcomes some of its drawbacks.

The Adadelta method modifies the learning rate of each parameter in a similar fashion to Adagrad, but instead of keeping all the prior gradients, it simply retains a moving average of the squared gradients. This helps to lower the memory needs of the method.

Additionally, Adadelta employs a method called "delta updates" to alter the learning rate. Instead of employing a set learning rate, Adadelta employs the ratio of the root mean squared (RMS) of the previous gradients and the RMS of the past updates to scale the learning rate. This helps to further prevent the learning rate from falling too soon for frequently occurring characteristics.

Like Adagrad, Adadelta is particularly effective for dealing with sparse data, but it may also perform better in cases where Adagrad may converge too rapidly.

AdaDelta is a stochastic optimization technique that allows for per-dimension learning rate method for SGD. It is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to a fixed size w .

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends only on the previous average and current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$

- $E[g^2]_t$: Running average of squared gradients at time t .
- γ : Decay factor or smoothing factor, a value between 0 and 1 that determines the weight given to the previous running average.
- $\gamma E[g^2]_{t-1}$: Running average of squared gradients at the previous time step ($t-1$).
- g_t : Gradient of the cost function with respect to the parameters at time t .

Usually γ is set to around 0.9. Rewriting SGD updates in terms of the parameter update vector:

$$\Delta\theta_t = -\eta * g_{t,i}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

AdaDelta takes the form:

$$\Delta\theta_t = \frac{-\eta g_t}{\sqrt{E[g^2]_t + \epsilon}}$$

- $\Delta\theta_t$: The parameter update at time t.
- η : Learning rate, a positive scalar determining the step size in the parameter space.
- g_t : Gradient of the cost function with respect to the parameters at t
- $E[g^2]_t$: Running average of squared gradients at time t
- ϵ : Small constant added for numerical stability to avoid division by zero in the denominator.

The main advantage of AdaDelta is that we do not need to set a default learning rate

Overall, Adadelta is a strong optimization technique that may assist expedite the training of deep neural networks and increase their performance, while resolving some of the drawbacks of Adagrad.

1.2.5.4 RMSProp (Root Mean Square Propagation)

RMSProp (Root Mean Square Propagation) is an optimization approach used in machine learning and deep learning to optimize the training of neural networks.

Like Adagrad and Adadelta, RMSProp modifies the learning rate of each parameter throughout the training process. However, instead of collecting all the prior gradients like Adagrad, RMSProp computes a moving average of the squared gradients. This helps the algorithm to modify the learning rate more slowly, and it prevents the learning rate from lowering too soon.

The RMSProp technique also utilizes a decay factor to limit the effect of prior gradients on the learning rate. This decay factor allows the algorithm to give more weight to recent gradients and less weight to older gradients.

One of the key advantages of RMSProp over Adagrad is that it can handle non-stationary goals, where the underlying function that the neural network is trying to mimic varies over time. In certain instances, Adagrad may converge too rapidly, while RMSProp may adjust the learning rate to the changing objective function.

Here's how RMSProp works:

- Initialization: Initialize a running average variable to zero $E[g^2]_t$, where g_t is the gradient of the cost function with respect to the parameters.
- Compute Gradients: At each iteration, compute the gradient g_t of the cost function with respect to the parameters.
- Update Running Average :

Update the running average using an exponential decay:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g^2_t$$

Here γ is a decay factor (typically close to 1) that determines the weight of the previous running average compared to the current gradient.

- Update Parameters:

$$\Delta\theta_{t+1} = \Delta\theta_t - \frac{\eta g_t}{\sqrt{E[g^2]_t + \epsilon}}$$

Here η is the learning rate, and ϵ is a small constant added for numerical stability.

1.2.5.5 Adam (Adaptive Moment Estimation)

Adam (Adaptive Moment Estimation) is an optimization technique used in machine learning and deep learning to optimize the training of neural networks.

Adam integrates the notions of both momentum and RMSProp. It maintains a moving average of the gradient's first and second moments, which represent the mean and variance of the gradients, respectively. The moving average of the initial moment, which is comparable to the momentum term in other optimization methods, assists the optimizer to continue advancing in the same direction even when the gradients get smaller. The moving average of the second moment, which is identical to the RMSProp term, assists the optimizer to adjust the learning rate for each parameter based on the variance of the gradients.

Adam also contains a bias correction phase to alter the moving averages since they are biased towards zero at the beginning of the optimization process. This helps to increase the optimization algorithm's performance in the early stages of training.

Adam is a popular optimization technique because of its ability to converge rapidly and manage noisy or sparse gradients. Additionally, it does not require manual setting of hyperparameters like the learning rate decay or momentum coefficient, making it easier to use than other optimization techniques.

How it works

- Initialization:
 - Initialize parameters $m_0 = 0$ (initial moment estimate), $m_v = 0$ (initial second raw moment estimate).
 - Set $t = 0$ (iteration counter).

- Choose hyperparameters: α (learning rate), β_1 (exponential decay rate for the first moment estimate), β_2 exponential decay rate for the second raw moment estimate), ϵ (small constant for numerical stability).
- Compute Gradients:
 - At each iteration t , compute the gradient g_t of the cost function with respect to the parameters.
- Update First Moment Estimate:
 - Update the first moment estimate (momentum term):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
- Update Second Raw Moment Estimate:
 - Update the second raw moment estimate:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (g_t)^2$$
- Bias Correction:
 - Correct the bias in the first and second moment estimates, which tend to be biased towards zero in the early iterations:

$$m_{t'} = \frac{m_t}{\beta_1^t}$$

$$v_{t'} = \frac{v_t}{\beta_2^t}$$
- Update Parameters:

$$\Delta\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_{t'} + \epsilon}} m_{t'}$$
 - Here θ represents the parameters being optimized.
- Repeat:
 - Increment the iteration counter t and repeat steps 2-6 until convergence or a specified number of iterations.

Overall, Adam is a strong optimization method that may assist accelerate the training of deep neural networks and increase their performance.

1.2.5.6 Adamax

Adamax is a variation of the Adam optimization method used in machine learning and deep learning to optimize the training of neural networks.

Like Adam, Adamax also keeps a moving average of the gradient's first and second moments.

However, instead of employing the second moment of the gradients as in Adam, Adamax employs the L-infinity norm of the gradients. This is effective in circumstances when the gradients are relatively sparse or have a very large variation.

The adoption of the L-infinity norm in Adamax makes it more stable than Adam when dealing with sparse gradients. Additionally, the removal of the second moment term allows for faster convergence and smaller memory needs.

Overall, Adamax is a strong optimization technique that may assist expedite the training of deep neural networks and enhance their performance, particularly in cases when the gradients are sparse or have a large variation.

1.3 Comparative Analysis

Optimizer	Pros	Cons
Feature Scaling	<ul style="list-style-type: none"> -Prevents the model from giving a higher weight to certain attributes compared to others. -Feature scaling helps to make Gradient Descent converge much faster. 	<ul style="list-style-type: none"> -It does not always guarantee better results.
Batch Gradient Descent	<ul style="list-style-type: none"> - Efficient updates - Stable convergence - Allows parallel processing 	<ul style="list-style-type: none"> - Risk of premature convergence - Complexity accumulating errors at epoch end - Requires entire dataset in memory, limiting scalability - Slow updates, especially with large datasets - Demands more computational power
Mini-Batch Gradient Descent	<ul style="list-style-type: none"> - Frequent model updates for robust convergence, avoiding local minima - More computationally efficient than stochastic gradient descent - Allows for efficient processing without having all data in memory 	<ul style="list-style-type: none"> - Requires setting hyperparameter (mini-batch size) - Accumulating error information over mini-batch samples - Can generate complex functions
Stochastic Gradient Descent (SGD)	<ul style="list-style-type: none"> -Instantly assess model performance and improvement rates. -Easy to understand and implement, especially for beginners. -Faster updates facilitate quicker issue discovery. -Avoids local minima with a noisy update process. 	<ul style="list-style-type: none"> -Computationally intensive, especially with large datasets. -Noisy updates can result in variance across epochs. -Noisy learning process may make it challenging to commit to the model's minimum error.

	<ul style="list-style-type: none"> -Faster and requires less computational power. -Suitable for larger datasets. 	
Stochastic Gradient Descent with Gradient Clipping	<ul style="list-style-type: none"> - Mitigates Exploding Gradients: Enhances stability. -Improved Convergence: Reliable improvement, especially with noisy data. -Maintains Model Stability: Reduces risk of destabilizing updates. -Enhanced Generalization: Contributes to better generalization. 	<ul style="list-style-type: none"> -Potential Information Loss: May lead to information loss. -Hyperparameter Tuning: Requires tuning the clipping threshold. -Algorithm Complexity: Adds training complexity. -Dependence on Threshold: Effectiveness depends on the chosen threshold.
Momentum	<ul style="list-style-type: none"> - Reduces oscillations in the training process. - Faster convergence for ill-conditioned problems. 	<ul style="list-style-type: none"> - Increases the complexity of the algorithm. - No specific guideline for selecting the mini-batch size. -Longer training times due to a potentially high number of epochs. -Possibility of overshooting the global minimum.
Nesterov Momentum	<ul style="list-style-type: none"> - Converges faster than classical momentum. - Can reduce overshooting. 	<ul style="list-style-type: none"> - More expensive than classical momentum.
Adagrad	<ul style="list-style-type: none"> - Adaptive learning rate per parameter. - Effective for sparse data. 	<ul style="list-style-type: none"> - Accumulation of squared gradients in the denominator can cause learning rates to shrink too quickly. - Can stop learning too early.
Adadelata	<ul style="list-style-type: none"> - Can adapt learning rates even more dynamically than Adagrad. - No learning rate hyperparameter. 	<ul style="list-style-type: none"> - The learning rate adaptation can be too aggressive, which leads to slow convergence.
RMSProp	<ul style="list-style-type: none"> - Adaptive learning rate per parameter that limits the accumulation of gradients. - Effective for non-stationary objectives. 	<ul style="list-style-type: none"> - Can have a slow convergence rate in some situations.
Adam	<ul style="list-style-type: none"> -Easy to implement. -Computationally efficient. -Small memory requirements. -Invariant to diagonal scale change of gradients. -Very suitable for problems that are large in terms of data and / or parameters. -Suitable for non-stationary targets. -Suitable for problems with very 	<ul style="list-style-type: none"> - Requires careful tuning of hyperparameters.

	noisy or sparse gradients. -Hyperparameters are intuitive to interpret and usually require little adjustment. -Adaptive learning rate and momentum for each parameter -Learning rate does not diminish as in AdaGrad	
Adamax	- More robust to high-dimensional spaces. - Performs well in the presence of noisy gradients.	- Computationally expensive.

Table 1 Pros and cons Optimizers

CHAPTER 2 – CONTINUAL LEARNING AND TEST PRODUCTION IN MACHINE LEARNING SOLUTIONS

2.1 Continual Learning.

2.1.1 Introduction.

Machine Learning is a sort of Artificial intelligence (AI) that allows computers the capacity to learn without being explicitly trained by people to do so. On a fundamental level, Machine Learning employs algorithms to offer computers the capacity to study data, detect patterns, and create prediction outputs. The difficulty, however, is that a standard Machine Learning model anticipates that the data would be comparable to the data that it was trained on, which isn't necessarily the case.

Continuous Machine Learning (CML) flips this issue on its head by monitoring and retraining models with current data. The purpose of CML is to emulate a human's ability to collect and fine-tune knowledge continually. In this essay, we will go into detail about what precisely CML is, the obstacles related with it, and why it's vital for the growth of AI.

Before we get into the details of Continuous Machine Learning, it's important to first define the DevOps technologies that it relies on – CI, CT, and CD.

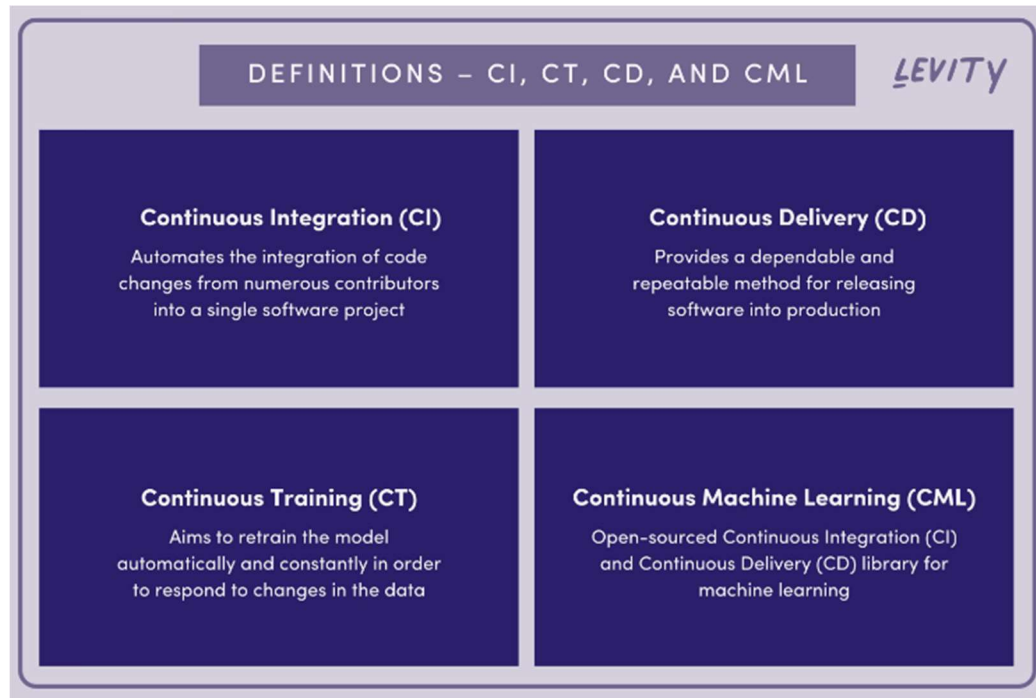


Figure 7: Definitions – CI, CT, CD, and CML

Continuous Integration (CI)

The practice of automating the integration of code changes from several contributors into a single software project is known as Continuous Integration (CI). It's a key DevOps best practice that allows developers to routinely consolidate code changes into a single hub where builds and tests are subsequently done. Before incorporating new code, automated tools are checked to confirm their correctness.

The CI approach relies significantly on a source code version control system. In addition, further checks, such as automated code quality tests, syntax style review tools, and others, are introduced to the version control system.

Continuous Training (CT)

As stated previously, Machine Learning (ML) models presume that data will always be comparable to the data that it was trained with. However, it is not always so. Namely, the majority of models work in situations where data is changing frequently and where "concept drifts" are likely to occur, which may have a detrimental influence on the accuracy and dependability of the models' predictions. To avoid "concept drifts" from developing, models need to be checked and retrained when the data gets too wrong.

This is where the notion of Continuous Training comes in. CT is a component of the MLOps practice paradigm. It seeks to retrain the model automatically and continually in order to adapt to

changes in the data. This practice avoids a model from becoming unreliable and incorrect.

Continuous Delivery (CD)

The purpose of Continuous Delivery is to create a dependable and repeatable mechanism for deploying software into production. Continuous Delivery for Machine Learning (CD4ML) expands this method by enabling a cross-functional team to construct Machine Learning applications based on code, data, and models that improve in tiny, safe increments that can be reproduced and reliably published at any time.

Continuous Machine Learning (CML)

Now that we have a good understanding of the principles above, we can look a bit closely at Continuous Machine Learning.

CML, short for Continuous Machine Learning, is an open-sourced Continuous Integration (CI) and Continuous Delivery (CD) library for Machine Learning. Generally speaking, it may be used to automate elements of your Machine Learning process, such as model training and assessment, comparing ML trials over your project history, and monitoring changes in datasets. It functions on the foundation of the MLOps ecosystem, which allows you to utilize your preferred DevOps tools on Machine Learning projects.

If you are acquainted with Netflix's recommender system, which features a "Up Next" feature that plays shows similar to the ones you've just viewed, then you have seen a CML model in action. In order to keep up with the never-ending supply of new programs, as well as changing interests and patterns of Netflix subscribers, the incoming data must be regularly inputted. And the model must continually update to have the capacity to propose appropriate shows or movies.

2.1.2 The Continuous Learning Process

Continuous learning represents a progression from conventional machine learning practices, incorporating familiar modeling principles such as pre-processing, model selection, hyperparameter tuning, training, deployment, and monitoring. To enhance adaptability to evolving data, two additional steps become essential in the continuous learning process: data rehearsal and the formulation of a continuous learning strategy. These steps are designed to optimize the model's assimilation of new data streams, aligning with the specific requirements and context of the data task.

The Continuous Learning Process

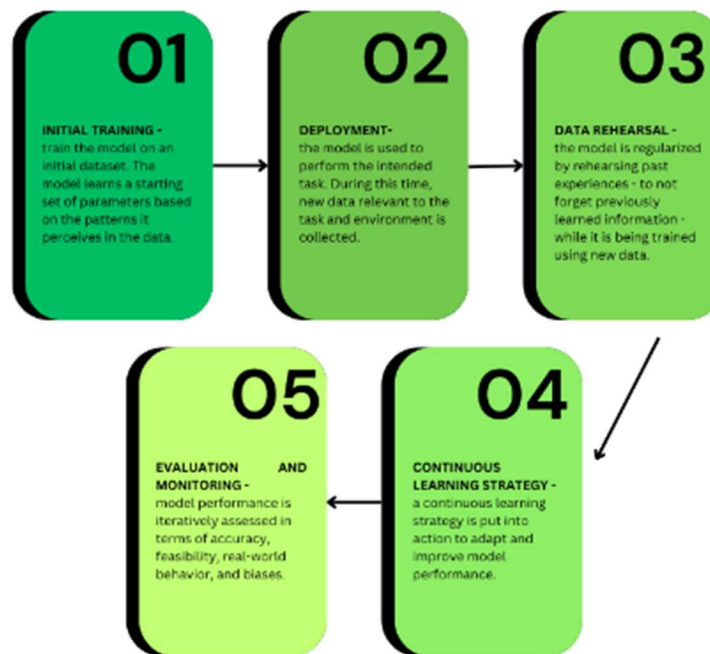


Figure 8: Flowchart of the Continuous Learning Process to Machine Learning

2.1.3 Types of continuous machine learning

1. Incremental Learning:

a. Dynamic Architectures:

Description: Utilize models with architectures that can adapt and expand to incorporate new knowledge without forgetting old tasks.

Application: Networks with dynamic nodes or adaptable structures that can grow incrementally.

b. Progressive Neural Networks (PNN):

Description: Allow for the addition of new units to the network to learn new tasks without affecting existing knowledge.

Application: Progressive expansion of neural networks over time.

c. Fine-tuning:

Description: Retrain the model on new tasks with a smaller learning rate to adapt to incremental changes.

Application: Updating the model parameters to accommodate new tasks while retaining knowledge from previous tasks.

2. Transfer Learning:

a. Feature Extraction:

Description: Use pre-trained models on a source task and transfer the knowledge by extracting features and training only the final layers on the target task.

Application: Apply knowledge gained from one domain to another related domain.

b. Domain Adaptation:

Description: Adapt a pre-trained model from one domain to perform effectively in a different but related domain.

Application: Transferring knowledge across domains with slight variations.

c. Multi-Task Learning:

Description: Simultaneously train a model on multiple tasks, leveraging shared representations.

Application: Learning a common feature space that benefits multiple tasks.

3. Lifelong Learning:

a. Memory-Augmented Networks:

Description: Integrate external memory to store and retrieve information learned over time.

Application: Preserve important knowledge from previous tasks in external memory.

b. Task-Aware Architectures:

Description: Design models that are aware of the tasks they are currently performing, allowing for task-specific adjustments.

Application: Dynamically adapt the architecture based on the current task.

c. Meta-Learning:

Description: Train a model to quickly adapt to new tasks based on experience gained from previous tasks.

Application: Rapid learning and adaptation to new tasks with minimal data.

4. Experience Replay Methods:

a. Replay Buffer:

Description: Store and randomly sample experiences from a buffer of past data during training.

Application: Mitigate catastrophic forgetting by periodically revisiting and training on past experiences.

b. Elastic Weight Consolidation (EWC):

Description: Regularize the model's parameters to protect important weights learned during previous tasks.

Application: Prevent forgetting by assigning higher penalties to crucial parameters.

5. Regularization Techniques:

a. EWC (Elastic Weight Consolidation):

Description: Introduce a penalty term in the loss function to preserve important weights.

Application: Protect crucial parameters during training on new tasks.

b. L2 Regularization:

Description: Add a penalty term to the loss function based on the squared magnitude of the weights.

Application: Control overfitting and maintain a more stable model.

c. Dropout:

Description: Randomly deactivate neurons during training to prevent co-adaptation of hidden units.

Application: Improve generalization by reducing reliance on specific neurons.

In Continual Learning, combining these methods judiciously is often necessary to address the unique challenges associated with learning over time without forgetting past knowledge. Researchers continue to explore novel techniques to enhance the capabilities of continual learning models.

2.1.4 How Continuous learning process work in machine learning

Continual learning, also known as lifelong learning or incremental learning, is an area of machine learning that focuses on the ability of a system to learn and adapt over time as new data becomes available. The goal is to enable a model to learn from a stream of data without forgetting the knowledge it has acquired from previous experiences. Here is a step-by-step explanation of how continual learning works in machine learning, along with some algorithms and equations:

1.Task Definition

- Define the tasks that the model needs to learn over time. Tasks could be different data distributions, classification problems, or any other learning objectives.

2.Initialization

- Initialize the model with some initial parameters. This is typically done using standard training on an initial dataset for the first task.

3.Task Training

- Train the model on the current task using the available data. Standard supervised learning algorithms such as gradient descent can be used here.

$$Loss_{task}(\theta) = \sum_{i=1}^N Loss(f(xi; \theta), yi)$$

- where θ represents the model parameters, N is the number of training examples, xi is an input example, yi is the corresponding label, and f is the model's prediction function.

4. Parameter update

- Update the model parameters to minimize the task-specific loss.

$$\theta_{new} = \theta - \eta \nabla_{\theta} Loss_{task}(\theta)$$

where η is the learning rate, and ∇_{θ} represents the gradient with respect to the model parameters.

5.Regularization Techniques

Apply regularization techniques to prevent catastrophic forgetting. Regularization methods, such as elastic weight consolidation (EWC) or synaptic intelligence (SI), can be used to penalize changes to important parameters learned in previous tasks.

6.Memory Replay

Store representative examples from previous tasks in a memory buffer and replay them during training on new tasks. This helps to retain information about past tasks and mitigate forgetting.

$$Loss_{task}(\theta) = \sum_{j=1}^M Loss(f(xi; \theta), yi)$$

where M is the number of examples in the memory buffer.

7.Task Switching

Repeat steps 3-6 for each task in the sequence. As new tasks are introduced, the model adapts its parameters to perform well on both new and previously learned tasks.

8.Evaluation

Periodically evaluate the model's performance on all tasks to ensure that it maintains a good level of accuracy on previously learned tasks while still learning new ones.

2.1.5 Advantages and limitations of continuous learning process

Advantages of Continuous Learning : Continuous learning proves valuable across a spectrum of data projects, encompassing descriptive, diagnostic, predictive, and prescriptive analytics. Its significance is particularly emphasized in scenarios with rapidly changing data. In

comparison to conventional machine learning approaches, the advantages include:

- **Enhanced Generalization:**
 - Continuous learning equips the model to exhibit increased robustness and accuracy when confronted with novel data.
 - Advantage: Improved ability to generalize across diverse and evolving scenarios.
- **Information Retention:**
 - Through the adoption of continuous learning strategies, the model takes into account prior knowledge acquired in previous iterations.
 - Advantage: The model can accumulate information over time, mitigating the risk of forgetting previously learned patterns.
- **Adaptability:**
 - Models employing continuous learning demonstrate adaptability to new knowledge, such as concept drift and emerging trends.
 - Advantage: Greater predictive capabilities over the long term, ensuring relevance in the face of dynamic data environments.

Limitations of Continuous Learning: While continuous learning offers effective adaptability to new data, its associated costs and modeling challenges must be carefully considered. The increased computational complexity may lead to higher economic costs, and issues such as model management and data drift pose additional challenges in the implementation of continuous learning approaches.

Cost Considerations and Modeling Drawbacks:

Continuous learning, while effective in adapting to new data, introduces certain cost considerations and modeling drawbacks. From a cost perspective, continuous learning approaches tend to be more computationally complex compared to traditional methods. This increased complexity arises from the continual adaptation of the model to new data, requiring additional investments in data, human resources, and computing infrastructure. The economic costs associated with these demands can be a significant consideration for organizations.

On the modeling front, there are drawbacks to continuous learning:

Firstly, the issue of model management arises. With each update based on new data, a new model is formed, potentially leading to a large number of models. This proliferation complicates the identification of the best-performing models and introduces challenges in terms of model governance

and evaluation.

Secondly, the challenge of data drift poses a risk to the effectiveness of continuous learning approaches. Continuous learning relies on processing substantial volumes of new data, and abrupt changes in feature distribution, known as data drift, can undermine the model's predictive capabilities. Successfully addressing data drift becomes a critical challenge, requiring the implementation of strategies to maintain model accuracy in the face of dynamic data environments.

2.1.6 Applications of Continuous Learning

Computer Vision: Continuous learning is used to train algorithms in image recognition and classification tasks, such as facial recognition and object detection.

Cybersecurity: Continuous learning approaches are employed to continuously monitor IT security systems and detect threats like phishing attempts, network intrusions, and spam, ensuring a proactive security posture.

Healthcare: Continuous learning is applied in healthcare to improve diagnostic workflows, especially in fields like oncology and radiology, where the dynamic nature of diseases requires constant learning and adaptation.

Robotics: Continuous learning techniques are utilized to enhance the adaptability and performance of robots. By continuously learning from new experiences and optimizing their actions, robots can operate more efficiently in changing environments.

Natural Language Processing: Continuous learning is employed in NLP applications like chatbots and virtual assistants, enabling them to improve their language understanding and response generation over time through ongoing interactions.

2.2 Test Production

2.2.1 Introduction

Machine learning tests may be categorized into two types: standard software tests and machine learning (ML) tests. Software tests evaluate the written logic, while ML tests evaluate the learnt logic.

ML tests may be subdivided into two distinct components: testing and assessment. We are acquainted with ML assessment, which involves training a model and assessing its performance on a validation set that has not been seen before. This evaluation is carried out using metrics like as accuracy and the Area under the Curve of the Receiver Operating Characteristic (AUC ROC), as well

as graphics such as the precision-recall curve

Conversely, ML testing entails evaluating the behavior of the model. Pre-training tests, which may be executed without taught parameters, verify the accuracy of our written logic. Is the categorization probability bounded within the range of 0 to 1, for instance? Post-training assessments assess the extent to which the acquired reasoning aligns with the intended outcome. On the Titanic dataset, it is reasonable to anticipate that females would have a greater likelihood of survival compared to men.

2.2.2 The testing process

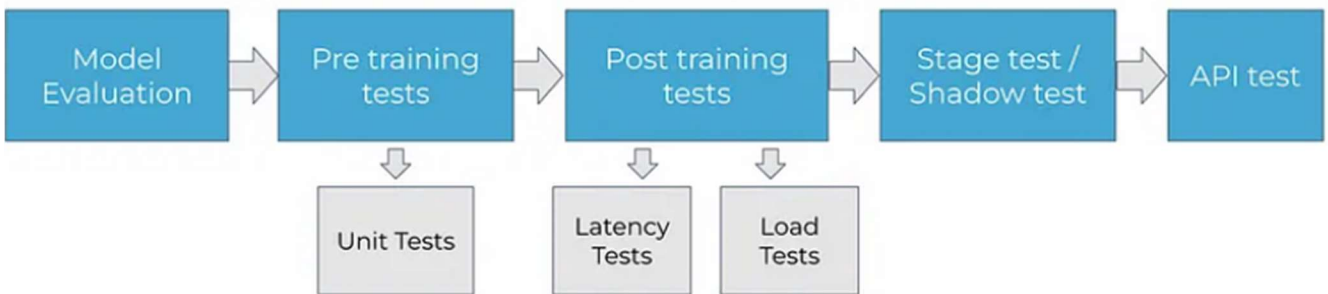


Figure 9: Machine learning testing flow

2.2.2.1 Model Evaluation

Model evaluation is stage 0 of model testing and is only limited to the functionality of the model. Below is the metric for each type of model to evaluate its quality. For imbalanced datasets, F1 scores, and AUC scores are best for classification and for outlier-heavy data, MAE is better for regression. For some models like the ensemble-decision tree-based supervised model (XGBoost, RF). SHAP can be used to understand what is the logic of prediction as such a black box model at the overall view and LIME for specific cases inspection

2.2.2.2 Pre-Training Test/ Unit Testing

Prior to the training process in both online and batch learning approaches, unit testing is conducted to ensure that the software/model meets the necessary criteria, such as the correct data format and sufficient data.

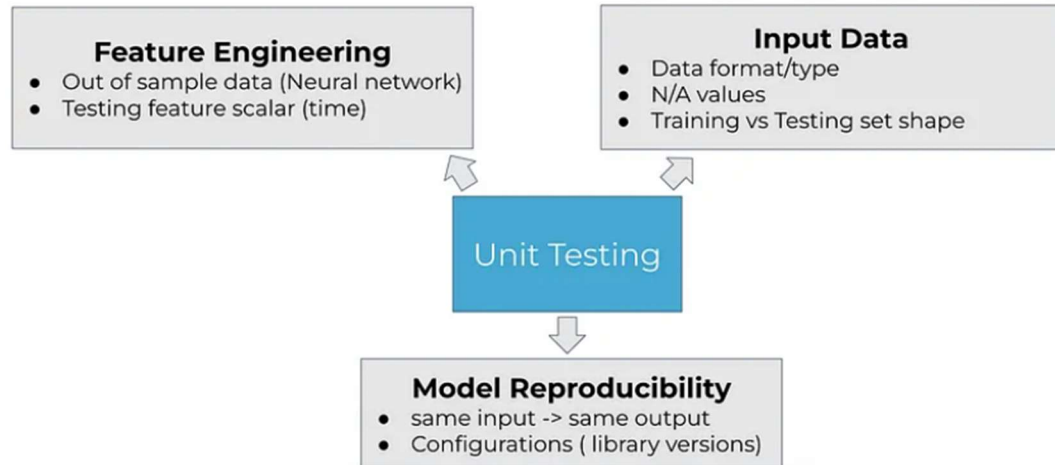


Figure 10: Unit testing

2.2.2.3 Post Training Test- Batch Vs Online Learning

The Post Training Test is conducted after the completion of training in batch learning, while it is performed during the training process for online learning.

Latency test: Check whether the prediction is produced within a fraction of a second so that the model can be scalable and manage the traffic. If it takes over a minute, the model design largely needs to be altered. This test is especially critical if it is an online machine-learning technique. One way to solve this in online learning is to define hyperparameters as static values and not utilize grid search cv to tweak the parameter every time new data arrives, since it may increase the latency. Therefore the static value of hyperparameters may be discovered by running sampled/subset dataset and training the model using approaches like random-search cv to acquire the optimal parameters based on majority validation data

Load test: Check how much test data can model handle at a time. This is critical for both batch and online learning. One approach to utilize SQLAlchemy to verify whether all DB (database) is accessed parallelly, this will allow handling of enormous loads of data. Also, the AWS container helps with CPU/memory monitoring. Locust is a tool to automate this test

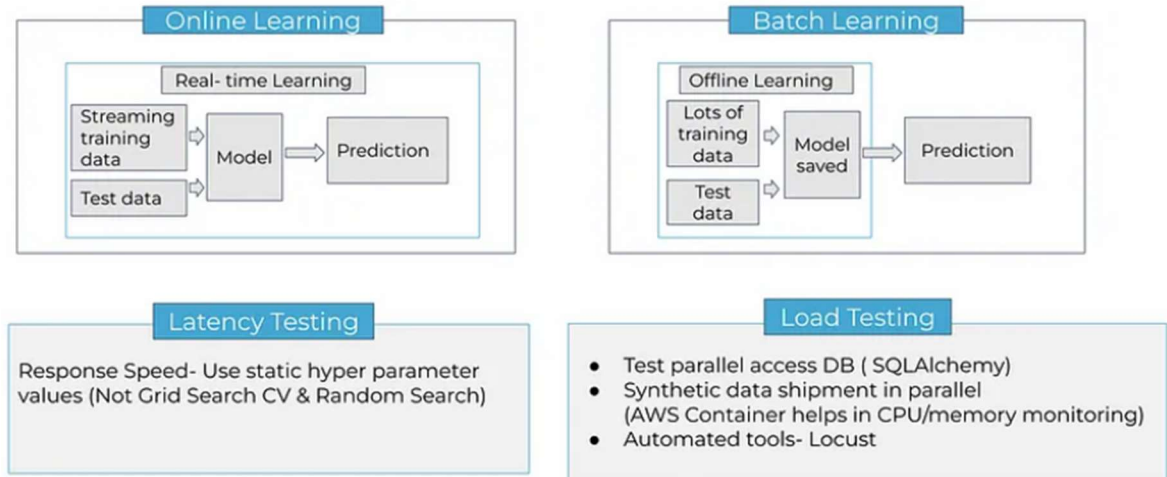


Figure 11: Post training test

2.2.2.4 A/B testing

With certain ML/AI models, over time, the data features change, hence the model trained on old data could not perform well with the new data. This is termed data drift. Similarly, idea drift happens when the assumptions created by a machine learning model no longer hold true in the real-world data it meets after deployment.

Therefore retraining is essential after ML implementation. A/B test helps determine if the new version of the model should replace the old one.

In A/B testing (automatically assisted by AWS Sagemaker) 80% of traffic is supplied to the current/old ml model while 20% of traffic (challenger) is provided to the new model. Based on baseline metric(s), the new model may replace the old one if the new performs better.

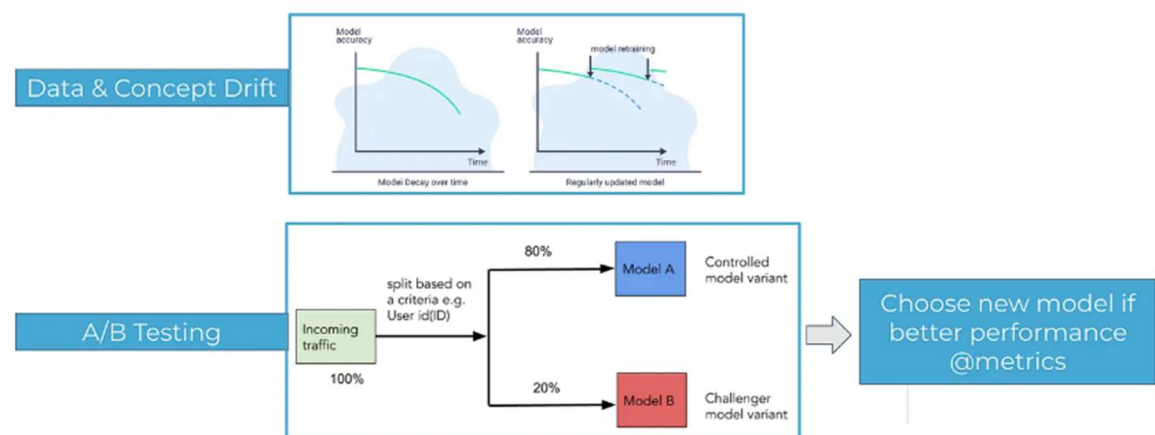


Figure 12: A/B testing

2.2.2.5 Stage test/ Shadow test

The Stage test is one of the last tests to determine whether the model will deliver the intended result. This test occurs after dockerization and AWS containerization for automated deployment in pipelines like Bitbucket and GitLab. This contains various test data (encompassing vast scenarios) that represent real-world data features and is delivered as input to the model at the stage environment.

Shadow test (Safe technique for a sanity check on big ML/AI models deployment):

- Deploy model in tandem with current model (if one is already there).
- For each request, route it to both models to create a forecast but only deliver the current one to the user
- Use the prediction on a new model for evaluation/analysis

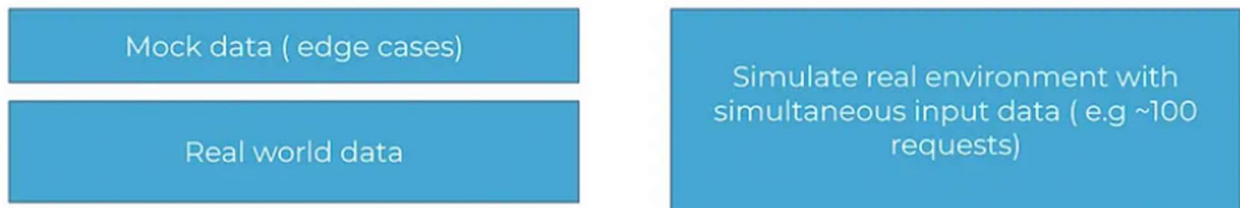


Figure 13: Stage test

2.2.2.6 API Testing

This is the final testing, where we verify how the real user will view the answer from the model. Therefore it catches all the various scenarios in user input that might generate problems in response. Each error is encoded as a unique Id (as status_code, error code). And the security aspect of input and output is tested for each unique consumer.



Figure 14: API Testing

2.2.3 Types of testing production

There are four primary kinds of tests which are employed at various phases in the development cycle:

- Unit tests: tests on discrete components that each have a single responsibility (ex. function that filters a list).
- Integration tests: tests on the integrated functioning of separate components (ex. data processing).

- System tests: tests on the architecture of a system for anticipated outputs given inputs (ex. training, inference, etc.).
- Acceptance tests: tests to verify that criteria have been satisfied, typically referred to as User Acceptance Testing (UAT).
- Regression tests: tests based on faults we've encountered previously to guarantee new modifications don't repeat them.

While ML systems are probabilistic in nature, they are made of numerous predictable components that may be verified in a similar fashion as conventional software systems. The contrast between testing ML systems comes when we shift from testing code to testing the data and models.

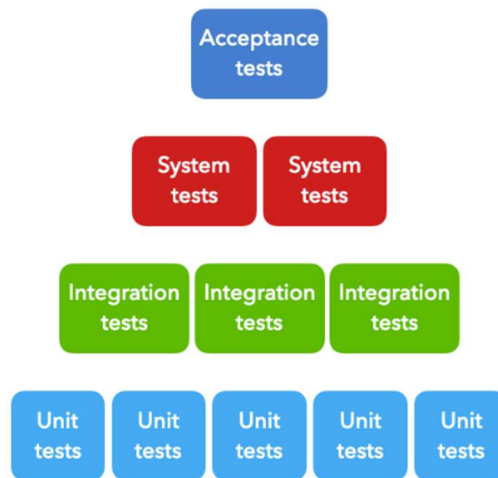


Figure 15: Type of tests

CHAPTER 3 – REFERENCES

1. Ben-David, S., & Shalev-Shwartz, S. (2018). Continual Learning in Neural Networks.
2. Chorowski, J., Cho, K., Bengio, Y., & Courville, A. (2021). Continual Learning for AI.
3. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning.
4. Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization Methods for Large-Scale Machine Learning.
5. Shanmugamani, R. (2017). Deep Learning for Computer Vision.
6. Kirkpatrick, J., et al. (2017). Overcoming Catastrophic Forgetting in Neural Networks.
7. Boyd, S., & Vandenberghe, L. (2004). Convex Optimization.
8. Jorgensen, P. C. (2013). Software Testing: A Craftsman's Approach.
9. Shalev-Shwartz, S., & Ben-David, S. (2014). Understanding Machine Learning: From Theory to Algorithms.