VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

FINAL REPORT

INTRODUCTION TO MACHINE LEARNING

Instructor: **Assoc.Prof.PhD. LE ANH CUONG**

Student: **BUI ANH PHU - 521H0508**

Class: **21H50302**

HO CHI MINH CITY, 2023

VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

FINAL REPORT

INTRODUCTION TO MACHINE LEARNING

Instructor: **Assoc.Prof.PhD. LE ANH CUONG**

Student: **BUI ANH PHU - 521H0508**

Class: **21H50302**

HO CHI MINH CITY, 2023

ACKNOWLEDGEMENT

To complete this essay, besides our own efforts, I have received a lot of help in terms of knowledge, experience, and skills from the school and teachers. First and foremost, we would like to express my special gratitude to Assoc.Prof.PhD Le Anh Cuong - the lecturer of Introduction to Machine Learning course who has taught us valuable knowledge of the subject. That knowledge is the foundation for me to continue learning and effectively apply it to this essay. Additionally, I would like to thank the teacher for allowing us to complete this essay, which has helped us to further develop my understanding of the subject. Thank you for your guidance and support in helping me to complete this essay to the best of my ability. Moreover, I would also like to express our gratitude to the school and the teachers who have compiled the Introduction to Machine Learning materials, providing me with useful resources for research and essay writing. Thank you sincerely!

COMPLETION OF THESIS AT TON DUC THANG UNIVERSITY

I here by certify that this thesis is my/our own work and was conducted under the guidance of Assoc.Prof.PhD Le Anh Cuong. The research and results presented in this thesis are truthful and have not been published previously in any form. The data presented in tables and figures used for analysis, comments, and evaluations were collected by the author from various sources and are clearly cited in the reference section.

Moreover, this thesis includes some comments, evaluations, and data from other authors and organizations, which are properly cited and referenced.

If any misconduct is detected, I fully take responsibility for the content of my thesis. Ton Duc Thang University is not liable for any copyright infringement that may occur during the thesis completion process.

Ho Chi Minh City, December 19, 2023

Author

(signature and full name)

ACKNOWLEDGEMENT AND EVALUATION SECTION BY INSTRUCTOR

Instructor's Acknowledgement Section

Ho Chi Minh City, 2023

(signature and full name)

Instructor's Evaluation Section

Ho Chi Minh City, 2023

(signature and full name)

SUMMARY

This essay provides a comprehensive overview of optimization algorithms in machine learning, primarily focusing on various gradient descent techniques. The first chapter delves into the fundamentals, explaining how optimizers work and detailing optimization algorithms such as Gradient Descent, Stochastic Gradient Descent (SGD), Mini Batch Stochastic Gradient Descent (MB-SGD), SGD with Momentum, Nesterov Accelerated Gradient (NAG), AdaGrad, AdaDelta, RMSprop, and Adaptive Moment Estimation (Adam). The chapter also discusses the importance of the learning rate and presents the advantages and disadvantages of each algorithm.

In the second chapter, the focus shifts to Continual Learning and Test in Production. Continual Learning is explored in depth, emphasizing its significance and introducing concepts like stateless retraining and stateful training. The challenges associated with Continual Learning, such as fresh data access, evaluation, data scaling, and algorithm challenges, are thoroughly examined. The essay categorizes Continual Learning into four stages, ranging from manual, stateless retraining to advanced continual learning.

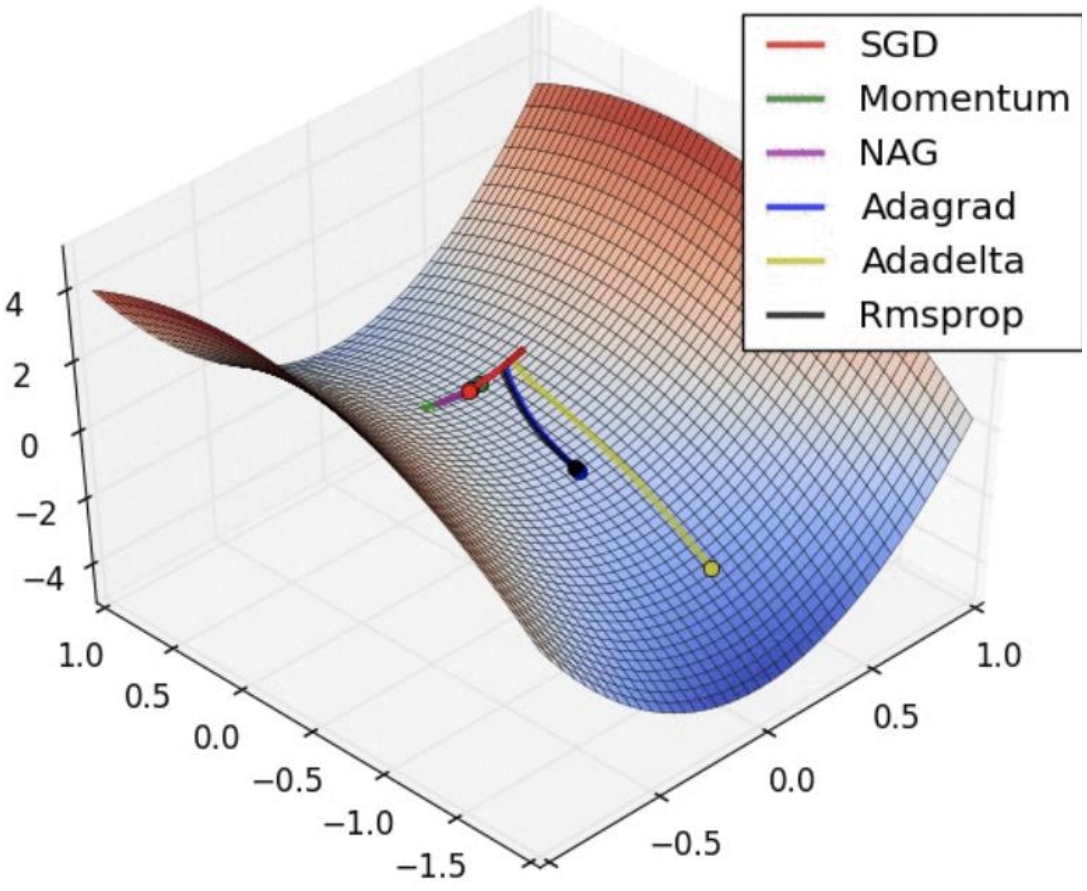
The second part of the second chapter concentrates on testing models in production, providing insights into pre-deployment offline evaluations. Various strategies for testing in production are discussed, including Shadow Deployment, A/B Testing, Canary Release, Interleaving Experiments, and Bandits. Each strategy is explained in detail, shedding light on its application and benefits.

In summary, the essay offers a comprehensive exploration of optimization algorithms in machine learning and delves into the challenges and stages of Continual Learning. Additionally, it provides valuable insights into testing models in production, presenting a range of strategies to ensure robust and effective deployment of machine learning models.

TABLE OF CONTENTS

CHAPTER 1: OPTIMIZATION ALGORITHMS IN MACHINE LEARNING

Overview of some of the most used optimizers while training a neural network.



1.1 Introduction

In the field of machine learning, the concept of loss quantifies the model's current performance, indicating how poorly it is performing. The objective is to leverage this loss information to enhance the model's capabilities. The primary goal is to minimize the loss, as a lower loss corresponds to improved model performance. The systematic procedure of reducing (or increasing) any mathematical expression is referred to as optimization.

Optimizers represent algorithms or techniques employed to adjust neural network attributes, such as weights and learning rate, with the aim of diminishing losses. These optimization methods are crucial for addressing optimization problems by minimizing the associated function.

1.2 How do Optimizers work?

For a helpful analogy, envision a hiker navigating down a mountain while blindfolded. Determining the exact direction is impossible, but she can discern whether she's progressing downward (making headway) or upward (losing ground). By consistently choosing paths leading downward, she eventually reaches the mountain base.

Similarly, establishing the optimal weights for your model right from the start is challenging. Yet, through trial and error guided by the loss function (analogous to the hiker's descent), you can progressively converge on the right configuration.

The adjustments to neural network weights or learning rates necessary for reducing losses are dictated by the optimizers employed. Optimization algorithms play a crucial role in minimizing losses and delivering the most precise outcomes.

Over the past few years, numerous optimizers have been researched, each carrying its own set of pros and cons. To gain a comprehensive understanding of these algorithms, explore the entire article, which delves into their workings, advantages, and disadvantages.

The article will delve into various types of optimizers, elucidating their mechanisms for precisely minimizing the loss function.

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini Batch Stochastic Gradient Descent (MB-SGD)
4. SGD with momentum
5. Nesterov Accelerated Gradient (NAG)
6. Adaptive Gradient (AdaGrad)
7. AdaDelta
8. RMSprop
9. Adam

1.3 Optimization algorithms

1.3.1 Gradient Descent

Gradient descent is an optimization algorithm utilized during the training of machine learning models. Operating on a convex function, it systematically adjusts its parameters through iterative steps to reduce a specified function towards its local minimum.

1.3.1.1 What is Gradient descent?

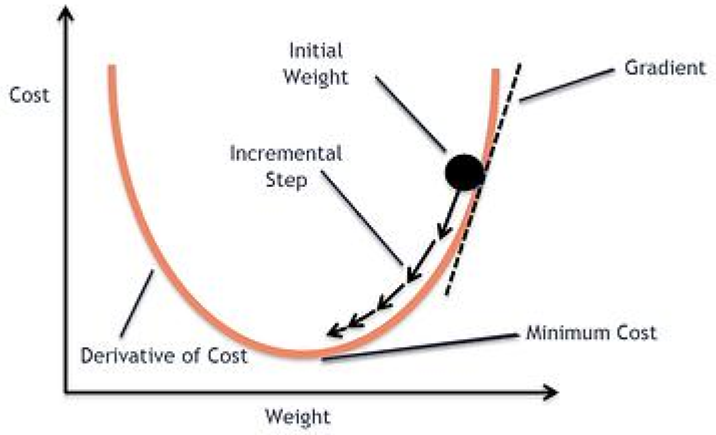
Gradient Descent stands as an optimization algorithm designed to locate a local minimum within a differentiable function. Its primary purpose is to identify the parameter values (coefficients) of a function that minimize a specified cost function.

The process commences by setting the initial values for the parameters, and then, through iterative steps guided by calculus, gradient descent systematically refines these values to minimize the given cost function.

Initialization strategies are employed to set the initial weight, and with each epoch, the weight undergoes updates in accordance with the prescribed update equation.

$$\text{Repeat until convergence } \{$$
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$
$$\}$$

The given equation calculates the gradient of the cost function J(θ) w.r.t to the parameters or weights θ across the entire training dataset:



Our objective is to reach the lowest point on our graph, representing the relationship between the cost and weights (Cost and weights) or a point where further descent is no longer possible a local minimum.

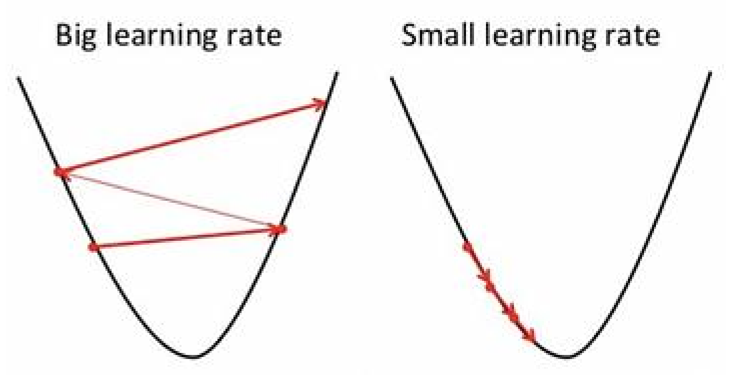
Now, let's explore the concept of "Gradient."

"A gradient measures how much the output of a function changes if you change the inputs a little bit." —Lex Fridman (MIT)

1.3.1.2 Importance of Learning rate

The size of the steps that gradient descent takes toward the local minimum is determined by the learning rate, which dictates the speed of our movement towards the optimal weights.

To ensure that gradient descent effectively reaches the local minimum, it is crucial to set the learning rate to an appropriate value neither too low nor too high. This consideration is significant because excessively large steps might result in the algorithm bouncing back and forth within the convex function of gradient descent (as illustrated in the left image below). Conversely, if the learning rate is set too small, gradient descent will eventually reach the local minimum, but the process may be prolonged (as depicted in the right image below).



Therefore, it is essential to avoid setting the learning rate either too high or too low. To assess the effectiveness of the learning rate, it can be visualized on a graph.

In programming code, the implementation of gradient descent typically resembles the following structure:

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

Over a predetermined number of epochs, the initial step involves computing the gradient vector, denoted as params_grad, of the loss function concerning our parameter vector, params, across the entire dataset.

1.3.1.3 Advantages and disadvantages

Advantages:

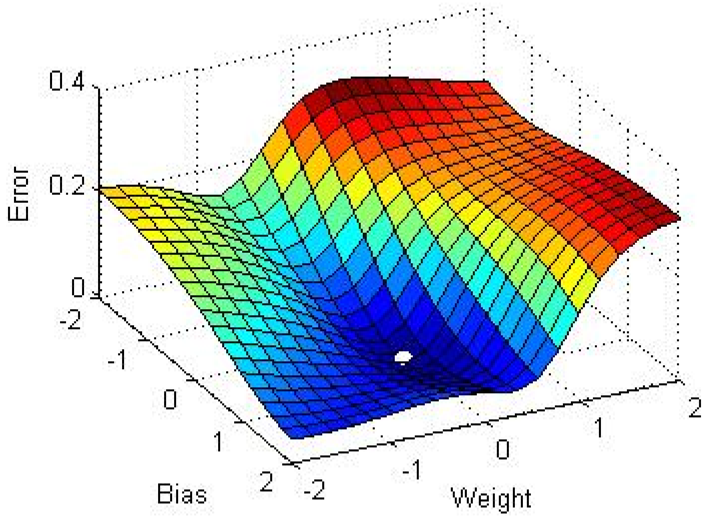
- Straightforward computation.
- Simple to implement.
- Easily understandable.

Disadvantages:

- Prone to getting stuck in local minima.
- Weight adjustments occur after calculating the gradient on the entire dataset. Consequently, if the dataset is extensive, convergence to the minima may take an extended period.
- Demands significant memory resources to calculate the gradient across the entire dataset.

1.3.2 Stochastic Gradient Descent (SGD)

The Stochastic Gradient Descent (SGD) algorithm is a development of the Gradient Descent, designed to address certain drawbacks of the GD algorithm. Gradient Descent is hindered by its need for substantial memory to process the entire dataset of n points simultaneously in order to compute the derivative of the loss function. In contrast, the SGD algorithm calculates the derivative by considering one point at a time, mitigating the memory requirements.



SGD conducts a parameter update for every training example, denoted as x(i), along with its corresponding label y(i):

$$\theta = \theta - \alpha \cdot \partial(J(\theta;x(i),y(i)))/\partial\theta$$

where {x(i) ,y(i)} are the training examples.

To expedite the training process, we perform a Gradient Descent step for each training example. The potential implications of this approach are illustrated in the image below.

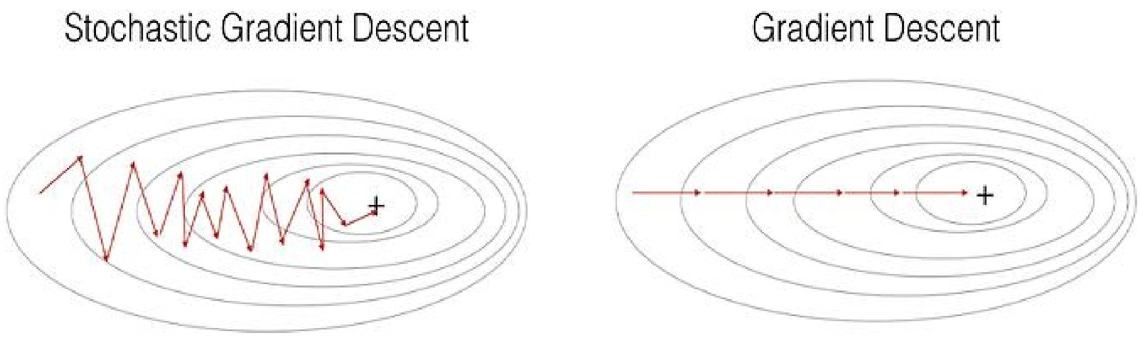


Figure 1 : SGD vs GD
"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

- On the left side, we observe Stochastic Gradient Descent (where m=1 per step), where a Gradient Descent step is taken for each individual example. On the right side is Gradient Descent (1 step per entire training set).

- SGD exhibits noticeable noise, yet it is considerably faster, albeit with a risk of not converging to a minimum.
- To strike a balance between the two approaches, Mini-batch Gradient Descent (MGD) is often employed. MGD involves examining a smaller subset of training set examples at a time, typically in batches of a certain size (commonly a power of 2, such as 2^6, etc.).
- Mini-batch Gradient Descent offers relative stability compared to Stochastic Gradient Descent (SGD), although it does introduce oscillations since gradient steps are taken based on a sample of the training set rather than the entire set, as in Batch Gradient Descent (BGD).

In SGD, it is noted that updates require more iterations compared to gradient descent to reach the minima. On the right side, Gradient Descent takes fewer steps to reach the minima, but the SGD algorithm introduces more noise and demands more iterations.

The code segment for SGD involves the addition of a loop over the training examples, where the gradient with respect to each example is evaluated.

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example,
params)
        params = params - learning_rate * params_grad
```

Advantage:

- Reduced memory requirements compared to the Gradient Descent (GD) algorithm, given that the derivative is computed using only one point at a time.

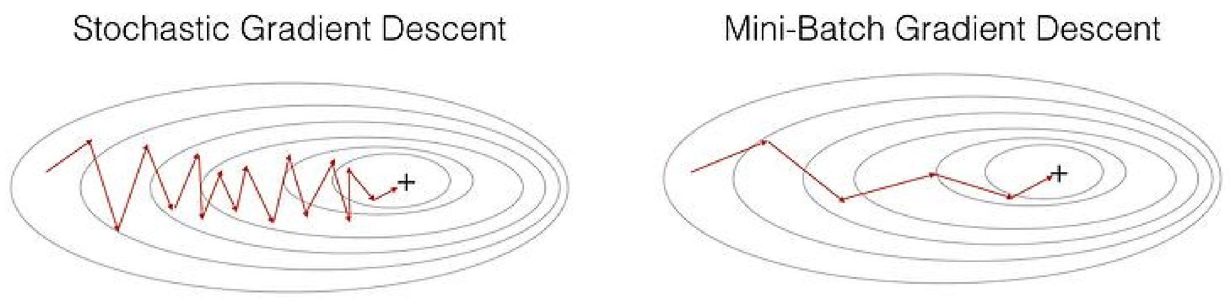
Disadvantages:

- Longer time is needed to complete one epoch compared to the GD algorithm.
- Slower convergence.
- Susceptible to getting stuck in local minima.

1.3.3 Mini Batch Stochastic Gradient Descent (MB-SGD)

MB-SGD algorithm extends the capabilities of the SGD algorithm, addressing the issue of high time complexity associated with SGD. MB-SGD, rather than considering one point at a time, takes a batch or subset of points from the dataset to compute derivatives.

It is observed that, after a certain number of iterations, the derivative of the loss function for MB-SGD closely resembles the derivative of the loss function for Gradient Descent. However, MB-SGD requires more iterations to reach the minima compared to GD, and the computational cost is also higher.



The weight update in MB-SGD relies on the derivative of the loss computed for a batch of points. The updates in the case of MB-SGD exhibit more noise because the derivative does not consistently point towards the minima.

MB-SGD partitions the dataset into several batches, and after processing each batch, the algorithm updates the parameters.

$$\theta = \theta - \alpha \cdot \partial(J(\theta; B(i)))/\partial\theta$$

where **{B(i)}** are the batches of training examples.

In the code, rather than iterating over individual examples, we now iterate over mini-batches, each containing 50 examples:

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size=50):
        params_grad = evaluate_gradient(loss_function, batch, 
params)
        params = params - learning_rate * params_grad
```

Advantage:

- Lower time complexity for convergence compared to the standard SGD algorithm.

Disadvantages:

- The updates in MB-SGD are more noisy compared to the updates in the GD algorithm.
- Longer time is required to converge compared to the GD algorithm.
- Susceptible to getting stuck in local minima.

1.3.4 SGD with Momentum

A significant drawback of the MB-SGD algorithm is the noisy updates in weight. This issue is addressed by SGD with Momentum, which mitigates the noise in gradients. The weight updates depend on noisy derivatives, and by denoising these derivatives, the convergence time can be reduced.

The concept involves denoising the derivative through exponential weighting averages, assigning more weight to recent updates compared to previous ones. This approach accelerates convergence in the relevant direction while minimizing fluctuations in irrelevant directions. An additional hyperparameter, referred to as momentum and denoted by **'γ'**, is introduced in this method.

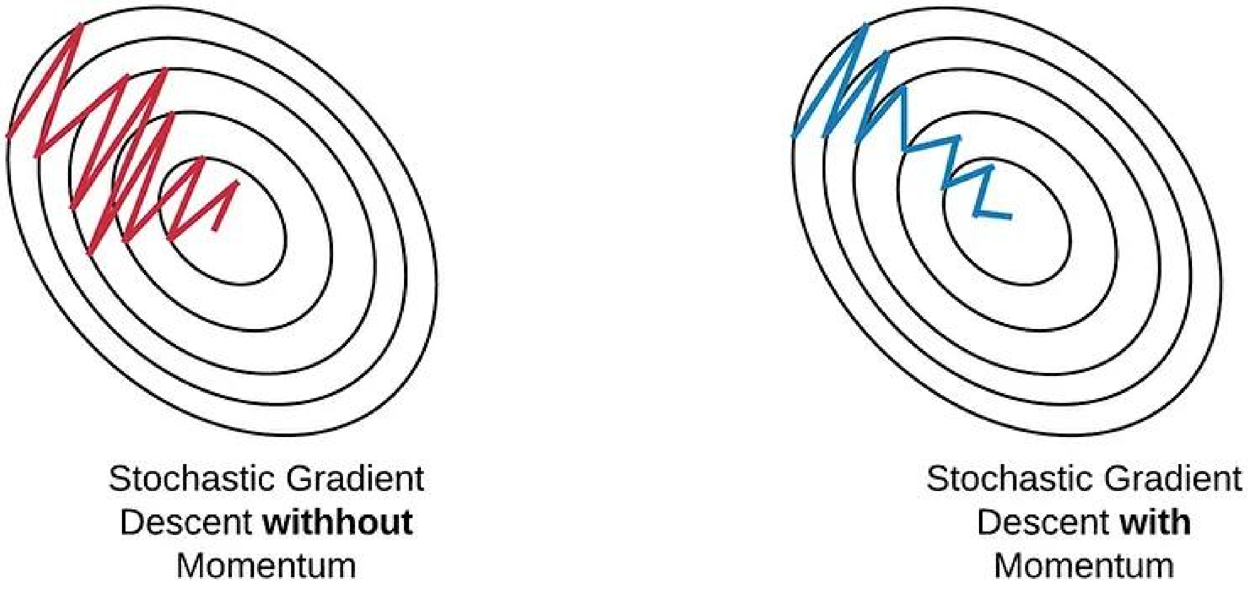
$$V(t) = \gamma . V(t-1) + \alpha . \partial(J(\theta))/\partial\theta$$

Now, the weights are updated by $\theta = \theta - V(t)$.

The typical choice for the momentum term, **γ**, is around 0.9 or a similar value.

Momentum at time 't' is calculated by considering all previous updates, assigning more importance to recent updates compared to previous ones. This strategy enhances the convergence speed.

In essence, when employing momentum, it's akin to rolling a ball down a hill. The ball accumulates momentum, gaining speed as it descends (until it reaches a terminal velocity if there is air resistance, i.e., when $\gamma<1$). A similar principle applies to our parameter updates: the momentum term increases for dimensions with gradients pointing in the same direction, reducing updates for dimensions with changing gradient directions. Consequently, this results in faster convergence and reduced oscillation.


Stochastic Gradient Descent **withhout** Momentum
Stochastic Gradient Descent **with** Momentum

The diagram depicted above demonstrates that SGD with momentum effectively denoises gradients, leading to faster convergence when compared to standard SGD.

Advantages:

- Retains all the advantages of the SGD algorithm.
- Achieves faster convergence than the GD algorithm.

Disadvantage:

- Requires the computation of an additional variable for each update.

1.3.5 Nesterov Accelerated Gradient (NAG)

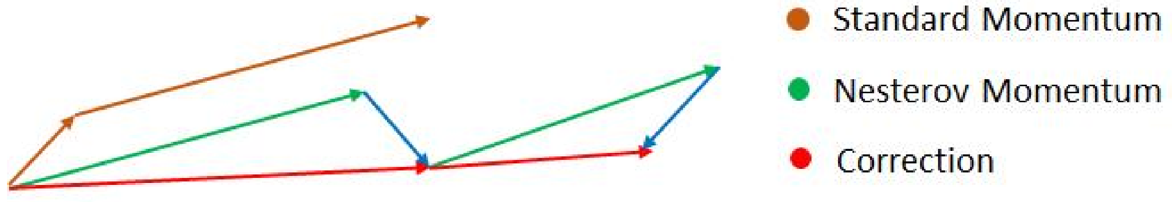
The concept behind the NAG algorithm is quite similar to SGD with momentum, with a subtle variation. In SGD with momentum, both momentum and gradient are computed based on the previously updated weights.

While momentum is a beneficial method, excessive momentum might cause the algorithm to overlook local minima and continue ascending. To address this issue, the NAG algorithm was introduced as a lookahead approach. By using **$\gamma.V(t-1)$** to adjust the weights, **$\theta - \gamma V(t-1)$** provides an approximate glimpse into the future location. Consequently, the algorithm calculates the cost based on this anticipated future parameter rather than the current one.

$$V(t) = \gamma . V(t-1) + \alpha . \partial(J(\theta - \gamma V(t-1)))/\partial\theta$$

and then update the parameters using **$\theta = \theta - V(t)$**.

Once again, the momentum term γ is typically set to a value around 0.9. While Momentum initially computes the current gradient (small brown vector in the Image below) and then takes a substantial step in the direction of the updated accumulated gradient (big brown vector), NAG follows a different sequence. NAG first takes a significant step in the direction of the previously accumulated gradient (green vector), evaluates the gradient, and then introduces a correction (red vector), culminating in the complete NAG update (red vector). This forward-looking update serves as a preventive measure to avoid excessive speed and enhances responsiveness, contributing significantly to the improved performance of RNNs across various tasks.



Both the NAG and SGD with momentum algorithms perform comparably well and exhibit the same set of advantages and disadvantages.

1.3.6 Adaptive Gradient Descent(AdaGrad)

In contrast to the previously discussed algorithms where the learning rate remains constant, AdaGrad introduces the concept of an adaptive learning rate for each weight. This approach involves making smaller updates for parameters linked to frequently occurring features and larger updates for parameters associated with infrequently occurring features.

For conciseness, the notation used includes gt to represent the gradient at time step t. **gt,i** as the partial derivative of the objective function w.r.t. with respect to the parameter **θi** at time step **t, η** as the learning rate, and $\nabla\theta$ as the partial derivative of the loss function **J(θi)**.

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

The SGD update for every parameter θ_i at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

In its update rule, Adagrad adjusts the general learning rate **η** at each time step **t** for each parameter **θi**, taking into account the historical gradients for **θi**:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

where **Gt** is the sum of the squares of the past gradients w.r.t to all parameters **θ.**

The advantage of AdaGrad lies in its ability to eliminate the necessity for manual tuning of the learning rate, with many practitioners opting for a default value of 0.01.

However, a notable weakness of AdaGrad is the accumulation of squared gradients (**Gt**) in the denominator. Due to the positivity of each added term, the accumulated sum continues to grow during training. This causes the learning rate to progressively shrink, eventually becoming exceedingly small and resulting in the vanishing gradient problem.

Advantage:

- Adaptive adjustment of the learning rate with iterations eliminates the need for manual updates.

Disadvantage:

- As the number of iterations increases, the learning rate decreases to an extremely small value, leading to slow convergence.

1.3.7 AdaDelta

The drawback with the previous algorithm, AdaGrad, was the diminishing learning rate with a high number of iterations, leading to sluggish convergence. To address this, the AdaDelta algorithm introduces the concept of taking an exponentially decaying average.

AdaDelta represents a more robust extension of Adagrad, adapting learning rates based on a moving window of gradient updates rather than accumulating all past gradients. This approach enables AdaDelta to continue learning effectively even after numerous updates. In the original version of AdaDelta, there is no need to set an initial learning rate.

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

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

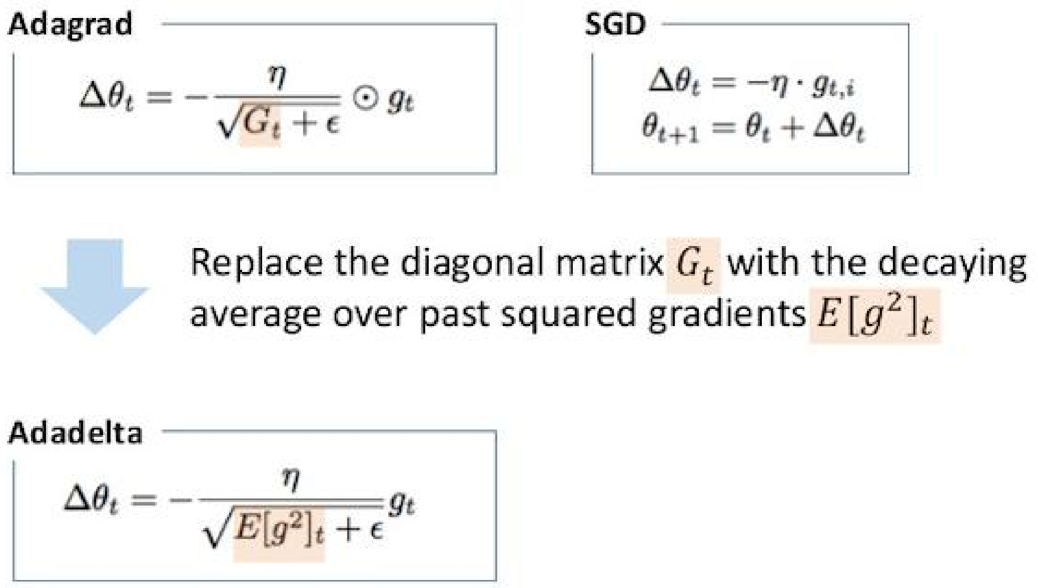
Usually γ is set to around 0.9. Rewriting SGD updates in terms of the parameter update vector:

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

AdaDelta takes the form:

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

With AdaDelta, there is no necessity to specify a default learning rate, as it has been removed from the update rule.


Adagrad
$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$
SGD
$\Delta\theta_t = -\eta \cdot g_{t,i}$
$\theta_{t+1} = \theta_t + \Delta\theta_t$
Replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$
Adadelta
$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$

1.3.8 RMSprop

RMSprop is essentially identical to the initial update vector of Adadelta that was derived earlier.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop also involves dividing the learning rate by an exponentially decaying average of squared gradients. Geoffrey Hinton recommends setting γ to 0.9, and a commonly suggested default value for the learning rate **η** is 0.001.

Both RMSprop and Adadelta were developed independently around the same time, driven by the shared objective of addressing the issue of Adagrad's rapidly diminishing learning rates.

1.3.9 Adaptive Moment Estimation (Adam)

Adam can be perceived as a fusion of RMSprop and Stochastic Gradient Descent with momentum.

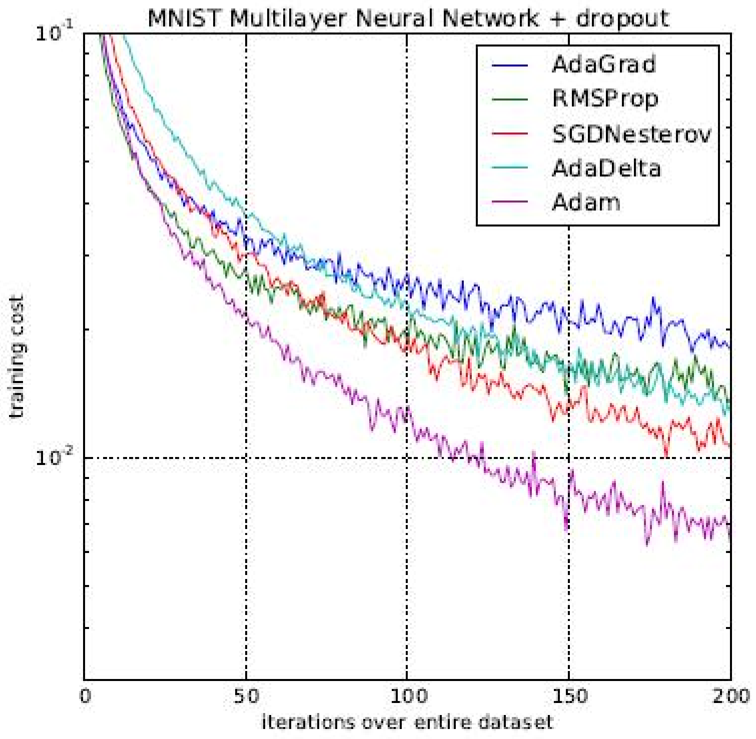
Adam calculates adaptive learning rates for each parameter. In addition to maintaining an exponentially decaying average of past squared gradients vt like Adadelta and RMSprop, Adam also retains an exponentially decaying average of past gradients mt, similar to the concept of momentum. While momentum resembles a ball rolling down a slope, Adam behaves akin to a heavy ball with friction, displaying a preference for flat minima in the error surface.

The hyperparameters β1 and β2, both belonging to the range [0, 1), govern the exponential decay rates of these moving averages. The computation of the decaying averages of past gradients mt and past squared gradients vt is performed as outlined below:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

The terms **mt** and **vt** represent estimations of the first moment (the mean) and the second moment (the uncentered variance) of the gradients, giving rise to the name of the method.

1.4 When to choose this algorithm?



As evident from the training cost, Adam consistently achieves the lowest values.

Now, considering the observations from the animation at the beginning of this article:

- SGD (red) appears to be stuck at a saddle point, indicating its limited applicability to shallow networks.

- All algorithms, excluding SGD, eventually converge. AdaDelta is the fastest, followed by momentum algorithms.
- AdaGrad and AdaDelta algorithms are suitable for sparse data.
- Momentum and NAG perform well across various cases but are relatively slower.
- Although the animation for Adam is not available, the plot suggests that it is the fastest algorithm to converge to the minima.
- Adam is regarded as the most effective algorithm among those discussed above.

CHAPTER 2: CONTINUAL LEARNING AND TEST IN PRODUCTION

This section addresses two significant and interconnected subjects: Continual Learning and Testing models in Production. The objective of exploring these topics concurrently is to acquire the skills needed to automate, ensure safety, and enhance the efficiency of updating models in a production setting.

2.1 Continual Learning

Continual Learning involves the concept of regularly updating your model as new data becomes accessible, allowing it to stay aligned with current data distributions.

However, once the model undergoes an update, it is not advisable to deploy it to production without thorough testing. It is crucial to conduct testing to verify the safety and superiority of the updated model compared to the existing one in production. This is the focus of the subsequent section, titled "Testing models in Production".

Continual learning is frequently misunderstood:

- Continual learning does NOT exclusively pertain to a specific category of ML algorithms designed for incremental model updates with each new datapoint, such as sequential Bayesian updating and KNN classifiers. These algorithms, often termed "online learning algorithms," represent a limited subset.
    - The concept of Continual learning is applicable to any supervised ML algorithm, not confined to a particular class.
- Continual learning does NOT involve initiating a retraining process every time a new data sample becomes available. In fact, this practice is risky, as it can render neural networks susceptible to catastrophic forgetting.

- Many companies that implement continual learning update their models in micro-batches, typically at intervals like every 512 or 1024 examples. The optimal number of examples varies depending on the specific task.

While Continual learning may initially appear to be a task for data scientists, it frequently demands substantial **infrastructure work** to be effectively implemented.

2.1.1 Why Continual Learning?

The fundamental purpose is to assist your model in **staying aligned with shifts in data distribution**. Several critical use cases underscore the need for swift adaptation to changing distributions, including:

- **Use Cases with Unpredictable and Rapid Changes**: Industries like ride-sharing face scenarios where unexpected and rapid changes occur. For instance, a concert in an unforeseen location on a random Monday could challenge the effectiveness of the "Monday pricing ML model".
- **Use Cases Lacking Training Data for Specific Events**: Certain situations, such as Black Friday or novel sale events in e-commerce, pose challenges in acquiring sufficient historical data for training. Adapting the model throughout the event becomes essential to predict user behavior.
- **Use Cases Prone to the Cold Start Problem**: The cold start problem arises when a model needs to make predictions for a new or logged-out user without any historical or outdated data. Adapting the model promptly upon receiving data from such users is crucial for providing relevant recommendations.

2.1.2 Concept: Stateless retraining VS Stateful training

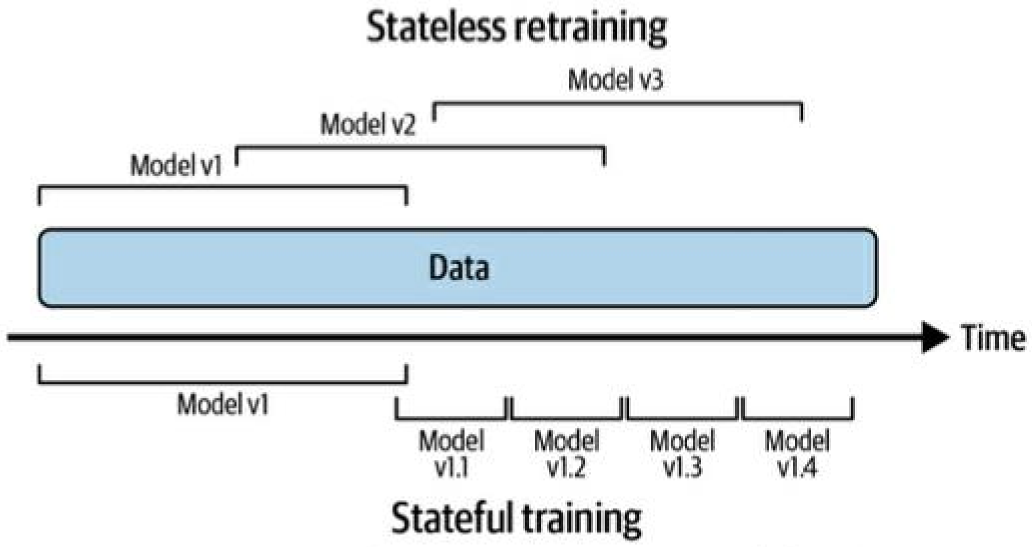


Figure 9-2. Stateless retraining versus stateful training

2.1.2.1 Stateless retraining

Initiate a complete retraining of your model on every occasion, utilizing freshly initialized weights and updated data.

- There could be instances of data overlap with what was previously used to train the prior version of the model.
- Many companies commence continual learning through a stateless retraining approach.

2.1.2.2 Stateful training (aka fine-tuning, incremental learning)

Initialize your model with weights from the previous training round and proceed with training on new, unseen data.

- This approach enables the model to update with considerably less data.
- Convergence is faster, and less compute power is required, with some companies reporting a 45% reduction.

- Theoretically, it may eliminate the need to store data once it's been used for training, addressing privacy concerns.
    - However, in practice, many companies tend to retain data even when unnecessary.
- Periodic **stateless retraining** with a large dataset is necessary to recalibrate the model.
- Once the infrastructure is properly configured, switching from stateless retraining to stateful training is a simple process.
- **Model iteration versus data iteration**: Stateful training is mainly employed to integrate new data **into an existing fixed model architecture** (i.e., data iteration). If you want to modify your model's features or architecture, a preliminary stateless retraining is required.
    - Some research exists on techniques like Net2Net knowledge transfer and model surgery for transferring weights between different model architectures. However, these methods have seen little to no adoption in industry so far.

2.1.3 Concept: feature reuse through log and wait

Features are computed for inference, and certain companies opt to store these calculated features for each data sample. This practice, known as **log and wait**, allows for the reuse of features in continual learning training, leading to computational savings. Additionally, it serves the purpose of aiding feature monitoring. While not widely adopted as of January 2023, log and wait is gradually gaining popularity.

2.1.4 Continual Learning Challenges

Despite its successful application in the industry, continual learning poses three significant challenges that companies must address.

2.1.4.1 Fresh data access challenge

To update your model every hour, obtaining **high-quality labeled** training data on an hourly basis is essential. The more frequent the update schedule, the more crucial this challenge becomes.

Problem: Speed of data deposit into data warehouses

Numerous companies retrieve their training data from data warehouses such as Snowflake or BigQuery. However, data from various sources is fed into the warehouse through distinct mechanisms and at varying rates.

For instance, data in the warehouse may come directly from real-time transport (events), while other portions may be sourced from daily or weekly ETLs that transfer data from other origins.

A common solution to address this challenge involves extracting data directly from real-time transport for training before it gets deposited in the warehouse. This approach is particularly effective when the real-time transport is linked to a feature store. However, there are hurdles to implementing this strategy:

- Some data, especially from external vendor systems beyond your control, may not flow through events. To ensure freshness in such cases, finding a method to capture changes on those systems using events, web-hooks, or polling APIs becomes necessary.
- In certain companies, batched ETLs perform extensive processing and data joining within the data warehouse to enhance utility. Shifting to a full real-time transport strategy requires devising a way to replicate the same processing on a continuous stream of data.

Problem: Speed of labelling

The rate at which new data can be labeled often serves as a bottleneck. Tasks with inherent labels and short feedback loops are the most suitable for continual learning, as a **shorter feedback loop** allows for faster labeling.

If obtaining natural labels within the required timeframe is challenging, alternative approaches such as weak supervision or semi-supervision techniques can be considered, albeit with the trade-off of potentially noisier labels. As a last resort, recurrent and rapid crowdsourcing for label annotation may be an option.

Another factor influencing labeling speed is the **label computation strategy**:

- Batch label computation involves periodic processing of data deposited into the data warehouse. The labeling speed is dependent on both the speed of data deposition and the frequency of label computation jobs.
- Similar to the aforementioned solution, a common method to expedite labeling is to compute labels directly from real-time transport (events). However, this streaming computation comes with its own set of challenges.

2.1.4.2 Evaluation Challenge

Incorporating continual learning as a practice introduces the risk of significant model failures. The higher the frequency of model updates, the greater the chances of encountering failures.

Furthermore, continual learning creates opportunities for coordinated adversarial attacks aimed at poisoning the models.

Consequently, rigorous testing of models before their deployment to a broader audience becomes crucial.

- Testing is a time-consuming process, serving as a potential constraint on achieving the fastest model update frequency.
- For instance, a new model designed for fraud detection may require approximately two weeks to accumulate sufficient traffic for a confident evaluation.

2.1.4.3 Data scaling challenge

Feature calculation typically involves scaling, which, in turn, necessitates access to global data statistics such as minimum, maximum, average, and variance.

In the case of stateful training, global statistics must account for both the previous data used to train the model and the new data being employed for updates. Managing global statistics in this context can be challenging.

A common approach to address this challenge is to calculate or approximate these statistics incrementally as new data is observed, as opposed to loading the entire dataset at training time for computation.

- An illustrative example of this technique is “Optimal Quantile Approximation in Streams”.
- Sklearn’s StandardScaler offers a *partial_fit* method that enables a feature scaler to be used with running statistics. However, the built-in methods are slow and have limitations in supporting a broad range of running statistics.

2.1.4.4 Algorithm challenge

This issue arises when certain types of algorithms are employed and there is a need for rapid updates, such as every hour.

The algorithms in question are those that, by design, depend on having access to the complete dataset for training. Examples include matrix-based, dimensionality reduction-based, and tree-based models. Unlike neural networks or other weight-based models that can be incrementally trained with new data, these types of models require the entire dataset.

- For instance, PCA dimensionality reduction cannot be performed incrementally; the full dataset is necessary.

This challenge becomes particularly pronounced when there is a need for very fast updates, and waiting for the algorithm to process the full dataset is not feasible.

While there are some variations of these models designed for incremental training, their adoption is not widespread. Hoeffding Trees and sub-variants are examples of such algorithms.

2.1.5 The Four Stages of Continual Learning

Typically, companies progress through four stages when transitioning to continual learning.

2.1.5.1 Stage 1: Manual, stateless retraining

Retraining of models occurs only under two conditions: (1) when the model's performance has significantly deteriorated, reaching a point where it becomes counterproductive, and (2) when the team has the available time and resources to carry out the update.

2.1.5.2 Stage 2: Fixed schedule automated stateless retraining

This stage typically occurs when the primary models in a domain have already been established, shifting the focus from creating new models to maintaining and enhancing existing ones. Remaining in stage 1 becomes too burdensome to overlook.

During this stage, the retraining frequency is often determined by intuition or a “gut feeling”.

The transition from stage 1 to stage 2 is commonly marked by the development of a script that someone creates to execute stateless retraining at regular intervals. Writing this script's complexity varies based on the number of dependencies that need coordination for model retraining.

The fundamental steps of this script include:

1. Retrieve data.
2. Perform downsampling or upsampling if required.
3. Extract features.
4. Process and/or annotate labels to generate training data.
5. Initiate the training process.
6. Assess the new model.
7. Deploy the updated model.

To implement this script, two additional components of infrastructure are necessary:

1. A scheduler.
2. A model store to automatically version and store all the artifacts required to reproduce the model. Established model stores include AWS SageMaker and Databrick's MLFlow.

2.1.5.3 Stage 3: Fixed schedule automated stateful training

To accomplish this, you must modify your script and establish a method for tracking data and model lineage. A straightforward model lineage versioning example includes:

- V1 vs V2, representing two distinct model architectures addressing the same problem.
- V1.2 vs V2.3 signifies that model architecture V1 is in its second iteration of a full stateless retraining, while V2 is in its third.
- V1.2.12 vs V2.3.43 indicates that there have been 12 stateful trainings performed on V1.2 and 43 on V2.3.
- It's likely that you'll need to combine this approach with other versioning techniques, such as data versioning, to maintain a comprehensive understanding of how the models are evolving.
- The author notes that there is no known model store with this type of model lineage capability, prompting companies to develop their own in-house solutions.
- At any given point, multiple models will be operational in production concurrently, facilitated by the arrangements outlined in Testing Models in Production.

2.1.5.4 Stage 4: Continual learning

In this stage, the **fixed schedule** component from previous stages is replaced by a **retraining trigger mechanism**. The triggers can be:

- **Time-based**.
- **Performance-based**, where retraining is triggered if performance falls below a certain threshold (e.g., below x% accuracy). However, measuring accuracy directly in production may not always be feasible, requiring the use of a weaker proxy.

- **Volume-based**, where retraining is prompted by a 5% increase in the total amount of labeled data.
- **Drift-based**, where retraining is initiated when a "major" shift in data distribution is detected. The challenge with drift-based triggers, as mentioned in the preceding chapter, is determining when such data distribution shifts become problematic for the model's performance degradation.

2.1.6 How often to Update your models

To address this question, you must initially comprehend and assess **the benefits derived from updating your model with fresh data**. The frequency of retraining should be proportionate to the extent of the gain achieved with each update.

2.1.6.1 Measuring the value of data freshness

An approach to measure the significance of fresher data is to train the same model architecture using data from three distinct time periods and subsequently evaluate each model with current labeled data (refer to the image below).

If the analysis reveals that allowing the model to stagnate for three months results in a 10% deviation in accuracy on current test data, and if a 10% deviation is deemed unacceptable, then the conclusion is that retraining should occur more frequently than every three months.

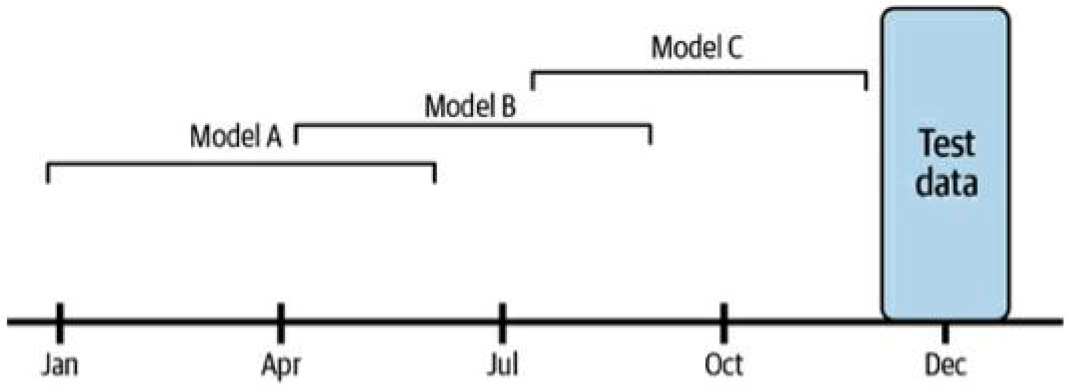


Figure 9-5. To get a sense of the performance gain you can get from fresher data, train your model on data from different time windows in the past and test on data from today to see how the performance changes

The illustration provides examples using multi-month datasets, but your specific use case might necessitate more detailed time buckets, such as weeks, days, or even hours.

2.1.6.2 When should I do model iteration?

Up to this point in the chapter, the focus has mainly been on updating your model with new data (i.e., data iteration). However, in practical scenarios, there may be instances where you need to modify the architecture of your model over time (i.e., model iteration). Here are some guidelines on when to consider model iteration and when to refrain:

- If consistently reducing the data iteration retraining trigger doesn't yield significant gains, it might be worthwhile to explore a better model (assuming it aligns with your business needs).
- In cases where transitioning to a larger model architecture requiring 100X compute power results in a marginal 1% performance improvement, while reducing the retraining trigger to 3 hours with 1X compute power also provides a 1% performance increase, prioritize data iteration over model iteration.

- The question of “when to do model iteration vs. data iteration” lacks a universally applicable answer across all tasks. Experimentation specific to your task is necessary to determine the optimal approach in each case.

2.2 Testing models in Production

To thoroughly assess your models before their broad deployment, a comprehensive strategy involves both **pre-deployment offline** evaluations and **testing in production** environment. Relying solely on offline evaluations is inadequate.

It is preferable for each team to establish a transparent pipeline outlining how models undergo evaluation. This includes specifying the tests to be conducted, designating responsible individuals, and establishing thresholds for promoting a model to the next stage. Automation of these evaluation pipelines, initiated with each new model update, is optimal. Similar to the evaluation of Continuous Integration/Continuous Deployment (CI/CD) in software engineering, reviews should be conducted for stage promotions.

2.2.1 Pre-deployment offline evaluations

The two most common methods are (1) using a **test split** for comparison against a baseline and (2) conducting **backtests**.

- **Test splits** are typically **static**, serving as a reliable benchmark for comparing multiple models. However, good performance on a static and older test split doesn't guarantee similar performance under the current data distribution conditions in production.
- **Backtesting** involves using the latest labeled data **that the model hasn't encountered during training** to assess performance (e.g., using data from the last hour if trained on data from the last day).
    - This summary introduces **backtests** for the first time.

- While it might be tempting to rely solely on backtesting to avoid testing in production, it is not sufficient. Production performance involves more than just label-related metrics. Factors like latency, user behavior with the model, and system integration correctness must be observed to ensure the model's safety for widespread deployment.

2.2.2 Testing in Production Strategies

2.2.2.1 Shadow Deployment

Intuition: Simultaneously deploy the new challenger model alongside the existing champion model. Route all incoming requests to both models, but serve only the predictions from the champion model. Record predictions from both models for subsequent comparison.

Pros:

- This method ensures the safest model deployment, as even if the new model has bugs, its predictions won't be served.
- It is a straightforward concept.
- This approach quickly accumulates enough data for statistically significant results since all models receive full traffic.

Cons:

- Not applicable when assessing model performance requires observing user interactions with predictions. For instance, predictions from shadow recommender models won't be served, making it impossible to determine if users would have clicked on them.
- It is cost-intensive as it doubles the number of predictions and, consequently, the required computing resources.
- If inference occurs using any online prediction modes, addressing edge cases such as variations in serving time or potential failure of the shadow model

becomes necessary. For instance, determining the response if the shadow model takes much longer than the primary model or if the shadow model fails.

2.2.2.2 A/B Testing

Intuition: Deploy the new challenger model alongside the existing champion model (model A) and direct a percentage of traffic to the challenger (model B). Present predictions from the challenger to users. Use monitoring and prediction analysis on both models to determine if the challenger's performance is statistically superior to the champion.

- For some use cases where dividing traffic is challenging, temporal splits can be employed, alternating between model A and model B on different days.
- The traffic split **must be a truly randomized experiment** to avoid selection bias, ensuring unbiased conclusions. For example, assigning model A to desktop users and model B to mobile users would introduce bias.
- The experiment duration should be sufficient to accumulate an adequate number of samples for statistically significant results.
- While statistical significance is not foolproof, if there is no discernible difference between A and B, either model can likely be used.
- Running A/B/C/D tests or more is feasible if desired.

Pros:

- Since predictions are served to users, this technique enables a comprehensive understanding of user reactions to different models.
- A/B testing is straightforward to comprehend, with ample libraries and documentation available.
- It is cost-effective as there is only one prediction per request.
- Edge cases arising from parallelizing inference requests in online prediction modes, as seen in shadow deployments, are avoided.

Cons:

- It is less secure than shadow deployments, as real traffic is exposed to the new model, requiring a stronger offline evaluation guarantee to prevent potential failures.
- A balance must be struck between assuming more risk (routing more traffic to the B model) and obtaining enough samples for a faster analysis.

2.2.2.3 Canary Release

Intuition: Deploy the challenger and champion models side by side, initially with the challenger receiving no traffic. Gradually transfer traffic from the champion to the challenger (the canary). Monitor the performance metrics of the challenger, and if they meet expectations, continue until all traffic is directed to the challenger.

- Canary releases can be combined with A/B testing for thorough performance measurement.
- Alternatively, canary releases can operate in "YOLO mode", where performance differences are visually assessed.
- Another version of a canary release involves introducing the challenger model to a smaller market first and then expanding to all markets if successful.
- If issues arise with the challenger model, traffic can be redirected to the champion.

Pros:

- Easy to understand.
- Simple to implement, especially if there is existing feature flagging infrastructure in the company.
- Since predictions from the challenger are served, this approach is suitable for models requiring user interaction to assess performance.
- Compared to shadow deployments, it is cost-effective, involving only one inference per request.

- When paired with A/B testing, it enables dynamic adjustment of the traffic distribution between models.

Cons:

- Possibility of being less rigorous in determining performance differences.
- If releases are not closely supervised, accidents can occur. While arguably less secure, it offers a straightforward rollback option.

2.2.2.4 Interleaving Experiments

Intuition: In A/B testing, a single user receives predictions exclusively from either model A or model B. In interleaving, a single user is provided with interleaved predictions from both model A and model B. User preferences for each model are measured, such as tracking which recommendations lead to more user clicks.

- Interleaving is commonly employed in recommendation tasks, but not all tasks are well-suited for this strategy.
- To prevent unfair user preference advantages, such as always favoring the top prediction from model A, equal likelihood should be ensured for the first slot to come from either model A or model B. The **team-drafting method** can be used for filling the remaining positions (refer to the image below).

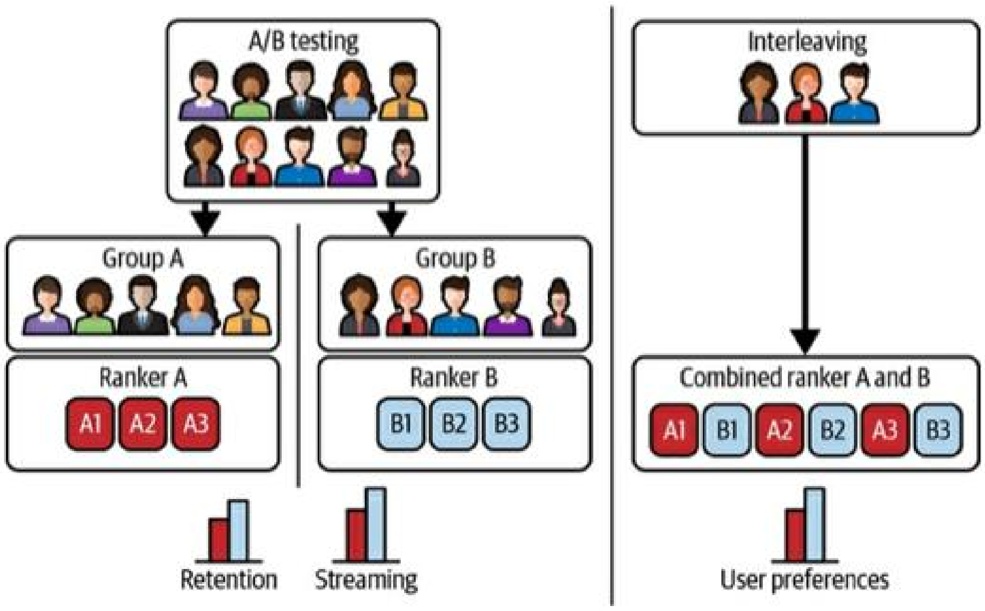


Figure 9-6. An illustration of interleaving versus A/B testing. Source: Adapted from an image by Parks et al.

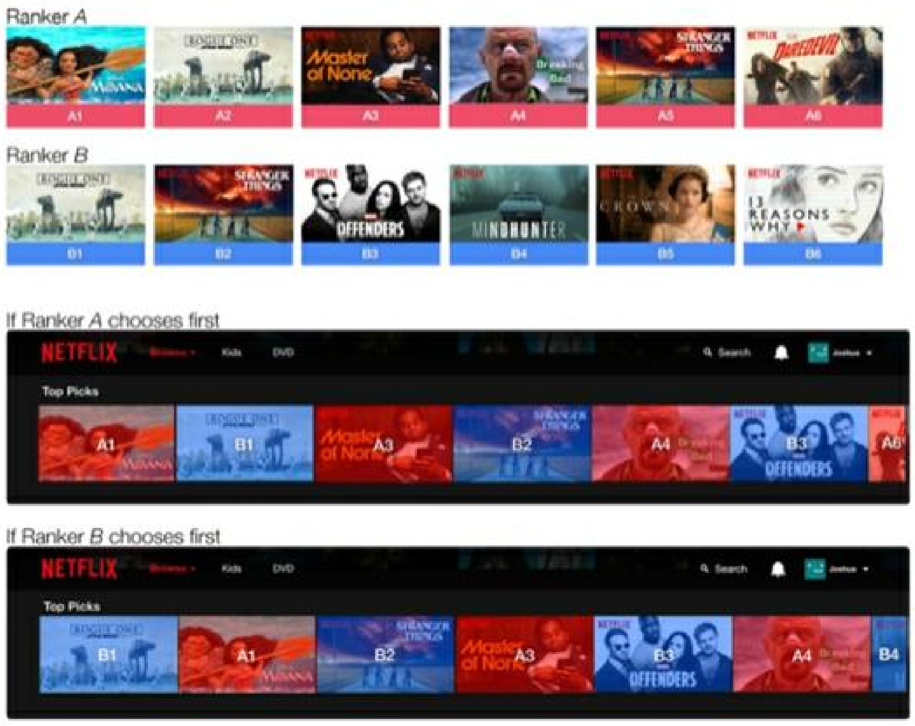


Figure 9-7. Interleaving video recommendations from two ranking algorithms using team draft. Source: Parks et al.[32]

Pros:

- Experimental findings from Netflix suggest that interleaving reliably identifies the best model with a **considerably smaller sample size** compared to traditional A/B testing, mainly because both models receive full traffic.
- Unlike shadow deployments, this strategy allows observation of user behavior against predictions since predictions are served.

Cons:

- Implementation is more complex than A/B testing.
    - Concerns about edge cases arise, such as determining the course of action if one of the interleaved models takes too long to respond or fails.
- It doubles the required compute power because each request receives predictions from multiple models.
- Not suitable for all types of tasks; for instance, it works well for ranking or recommendation tasks but may not make sense for regression tasks.
- Scaling to a large number of challenger models is challenging; the sweet spot seems to be 2-3 interleaved models.

2.2.2.5 Bandits

Intuition: Bandits are algorithms that assess the current performance of each model variant and dynamically decide, for each request, whether to exploit the current best-performing model (i.e., exploit current knowledge) or to explore other models to gather more information about them (i.e., explore to identify potentially better models).

- Bandits introduce the concept of **opportunity cost** to the decision-making process.
- Applicability of bandits depends on specific conditions. For bandits to be relevant, the use case must involve online predictions, as batched offline predictions are not compatible. Additionally, short feedback loops are required

to determine prediction quality swiftly and a mechanism to propagate feedback into the bandit algorithm for updating each model's payoff.

- Numerous bandit algorithms exist, with *epsilon-greedy* being the simplest, and *Thompson Sampling* and *Upper Confidence Bound (UCB)* being among the most powerful and popular.

Pros:

- Bandits require significantly less data than A/B testing to determine the superior model. For instance, a book example cites 630K samples for 95% confidence in A/B testing compared to 12K with bandits.
- Bandits are more data-efficient while simultaneously minimizing **opportunity cost**, making them optimal in many cases.
- In comparison to A/B testing, bandits are considered safer because they select poorly performing models less frequently. Convergence is faster, allowing swift elimination of inferior challengers.

Cons:

- Implementation of bandits is more challenging compared to other strategies due to the continuous need to propagate feedback into the algorithm.
- Applicability is limited to specific use cases (as outlined above).
- Bandits are not as secure as Shadow Deployments, as challengers receive live traffic.

REFERENCES

1. *Optimization algorithms in neural networks*. (2022, June 7). The AI Dream. https://www.theaidream.com/post/optimization-algorithms-in-neural-networks
2. Kumar, S. (2021, December 14). Overview of various Optimizers in Neural Networks - Towards Data Science. *Medium*. https://towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5
3. Doshi, S. (2021, December 7). Various optimization algorithms for training neural network. *Medium*. https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6
4. Bushaev, V. (2018, June 21). Stochastic Gradient Descent with momentum - Towards Data Science. *Medium*. https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d
5. Ruder, S. (2020, March 20). *An overview of gradient descent optimization algorithms*. ruder.io. https://ruder.io/optimizing-gradient-descent/
6. Huyen, C. (2022, Chapter 9). *Designing machine learning systems: An Iterative Process for Production-Ready Applications*. O'Reilly Media.