VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

FINAL PROJECT

MACHINE LEARNING

Instructor: **GV LE ANH CUONG**

Student's name: **TA DINH NAM – 519V0052**

Class: **19H50301**

Course: **23-24**

HO CHI MINH CITY, 2023

VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

FINAL PROJECT

MACHINE LEARNING

Instructor: **GV LE ANH CUONG**

Student's name: **TA DINH NAM – 519V0052**

Class: **19H50301**

Course: **23-24**

HO CHI MINH CITY, 2023

ACKNOWLEDGEMENT

Enterprise resource planning is an interesting, and useful subject, we would like to thank GV Le Anh Cuong guided and created the best conditions for us to complete this the final project.

With the deepest gratitude, we would like to send to the teachers at the Faculty of Information Technology who have imparted valuable knowledge to us during the study period. Thanks to the guidance and dedicated guidance of teachers, our research work can be completed well.

And thank Ton Duc Thang University for giving us a modern and developed educational environment.

With challenging work and effort, we have successfully completed this report. But surely, this report cannot avoid mistakes. We are looking forward to receiving from teacher so that we can improve it better.

We thank you!

THE PROJECT WAS COMPLETED

AT TON DUC THANG UNIVERSITY

I pledge that this is a product of our own project and is under the guidance of GV.Le Anh Cuong. The content of research, results in this subject is honest and not published in any form before. The data in the tables used for the analysis, comment, and evaluation were collect by the authors themselves from various sources indicated in the reference section.

In addition, the project also uses a few comments, assessments as well as data of other authors, other agencies, and organizations, with citations and source annotations.

If we find any fraud, we will take full responsibility for the content of my project. Ton Duc Thang University is not related to copyright and copyright violations caused by us during the implementation process (if any).

Ho Chi Minh City, December 4^{th}*, 2022*

Author

(Sign and provide full name)

Ta Dinh Nam

EVALUATION OF INSTRUCTING LECTURER

Confirmation of the instructor

_____

_____

_____

_____

_____

_____

Ho Chi Minh City, 2022

(Sign and provide full name)

The assessment of the teacher marked

_____

_____

_____

_____

_____

_____

Ho Chi Minh City, 2022

(Sign and provide full name)

TABLE OF CONTENTS

CHAPTER 1 – OPTIMIZER METHODS

1.1. What is an Optimizer in Machine Learning ?

In the field of machine learning, an optimizer is a critical component used to adjust the parameters of a model in order to minimize the error or maximize the performance of a learning algorithm. It plays a crucial role in the training process by optimizing the weights and biases of the model based on the input data.

An optimizer aims to identify the most effective set of parameters that minimizes the loss function, measuring the disparity between the model's predicted values and the true values. Through iterative parameter adjustments, the optimizer guides the model toward convergence, achieving minimal loss and accurate predictions. Viewing the optimization process as a journey across a intricate terrain of peaks and valleys, each peak signifies a distinct parameter configuration. The optimizer's role is to pinpoint the global or local minimum, signifying the optimal model configuration.

There are various optimization algorithms available, each with its own strengths and weaknesses. These algorithms are designed to efficiently search the parameter space and converge to an optimal solution. Gradient-based optimizers, such as Gradient Descent and its variants, are commonly used in machine learning due to their simplicity and effectiveness.

In the realm of training machine learning models, an optimizer holds a pivotal position. It refines the model's parameters, aiming to minimize errors and enhance overall performance. Through the application of diverse optimization algorithms, researchers and practitioners can address a broad spectrum of machine learning challenges, ultimately attaining cutting-edge results.

1.2. Popular Optimizers in Machine Learning

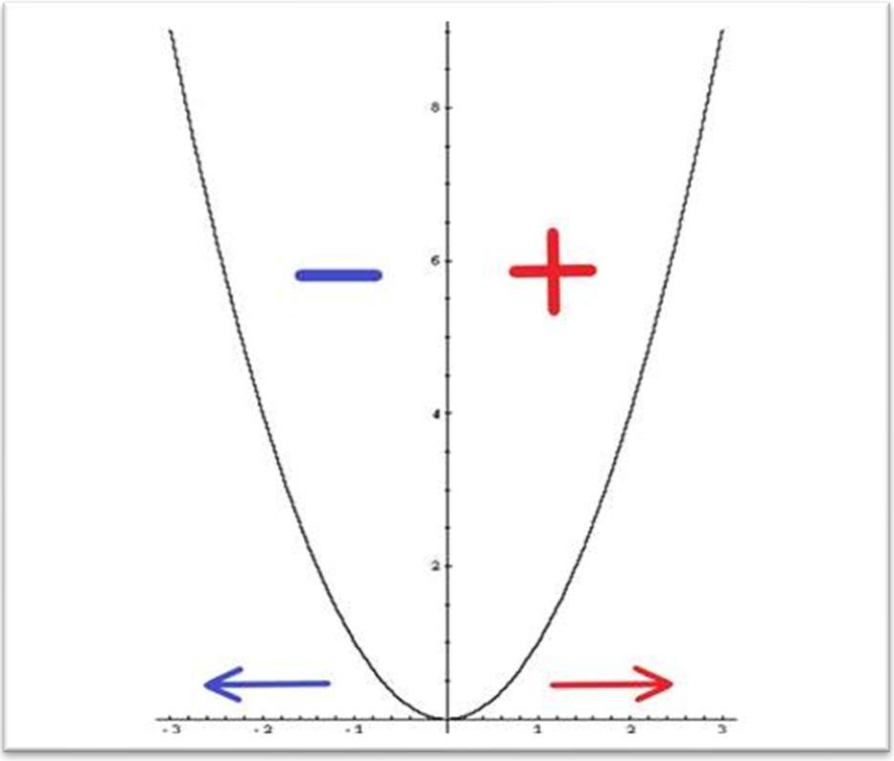
In the domain of machine learning, various widely employed optimizers play a crucial role in training models and enhancing their performance. These optimizers exhibit diverse approaches and strategies to fine-tune the model parameters. The following are some of the frequently utilized optimizers in the field:

- **Gradient Descent:** Gradient Descent stands out as a extensively employed optimization algorithm, modifying parameters in the direction opposing the loss function gradient. Commencing with randomly assigned initial parameter values, it continually updates them according to computed gradients. Gradient Descent encompasses various variants, including Batch Gradient Descent, Mini-Batch Gradient Descent, and Stochastic Gradient Descent (SGD).
- **RMSporop:** RMSprop, an optimization algorithm, dynamically adjusts the learning rate for individual parameters by considering the moving average of squared gradients. It incorporates a division of the learning rate by an ongoing average of the historical squared gradients, enhancing convergence speed. RMSprop demonstrates notable efficacy, particularly in addressing scenarios involving sparse data or non-stationary objectives.
- **Adam:** an acronym for Adaptive Moment Estimation, stands as an adaptive learning rate optimization algorithm that integrates the approaches of both RMSprop and Momentum. It dynamically adjusts the learning rate for each parameter by considering both the first-order moments (the mean) and the second-order moments (the uncentered variance) of the gradients. Adam is renowned for its favorable

convergence characteristics and resilience in handling gradients that are noisy or sparse.

- **Adagrad:** Integrate information about the frequency of occurrence of parameters in the past to adjust the learning coefficient.
- **Stochastic Gradient Descent (SGD):** Là phương pháp tối ưu cơ bản nhất, SGD cập nhật trọng số của mô hình bằng cách di chuyển theo hướng ngược của đạo hàm riêng của hàm mất mát.

1.3. Gradient Descent (GD)



In optimization problems, we often find the smallest value of a certain function, which reaches the minimum value when the derivative is 0. But the derivative of a function is not always possible, for For functions with many variables, the derivative is very complicated, even impossible. So instead, people find the point closest to the minimum point and consider that as the solution to the

problem. Gradient Descent translates into Vietnamese as gradually decreasing the gradient, so the approach here is to choose a random solution after each loop (or epoch) and let it gradually approach the desired point.

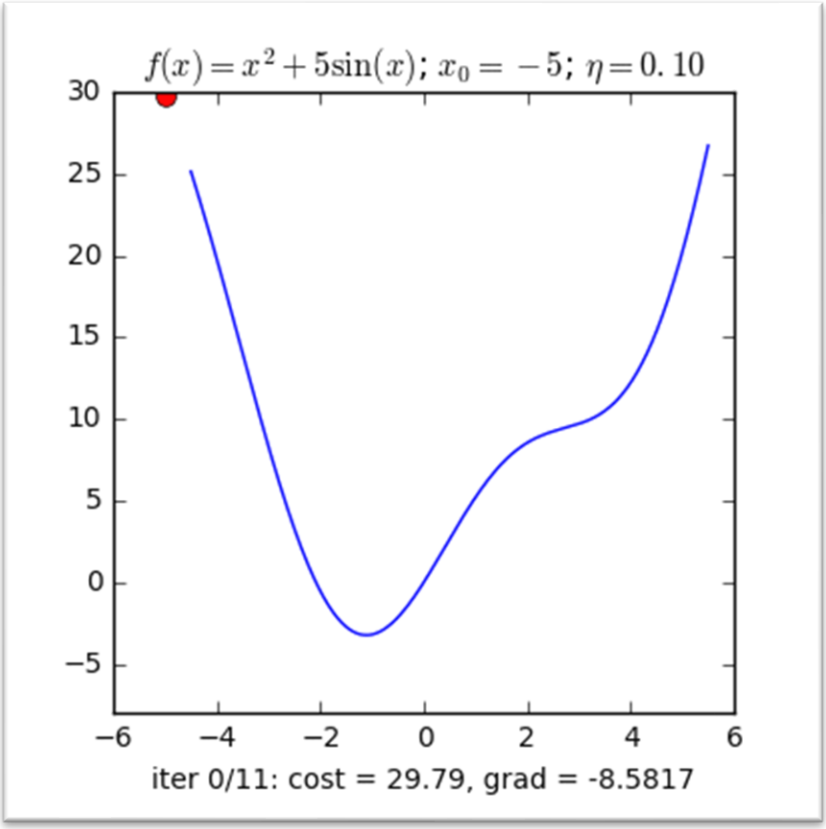
Formula: x(new) = x(old) - learningrate.gradient(x)

Ask the question why is there that formula? The above formula is built to update the solution after each iteration. The minus sign '-' here indicates the opposite direction of the derivative. Next question, why is the direction of the derivative opposite?

For example, for the function f(x)= 2x +5sin(x) as shown below, f'(x) =2x + 5cos(x) with x_old =-4 then f'(-4) <0 => x_new > x_old so the solution will move to the right closer to the minimum point.

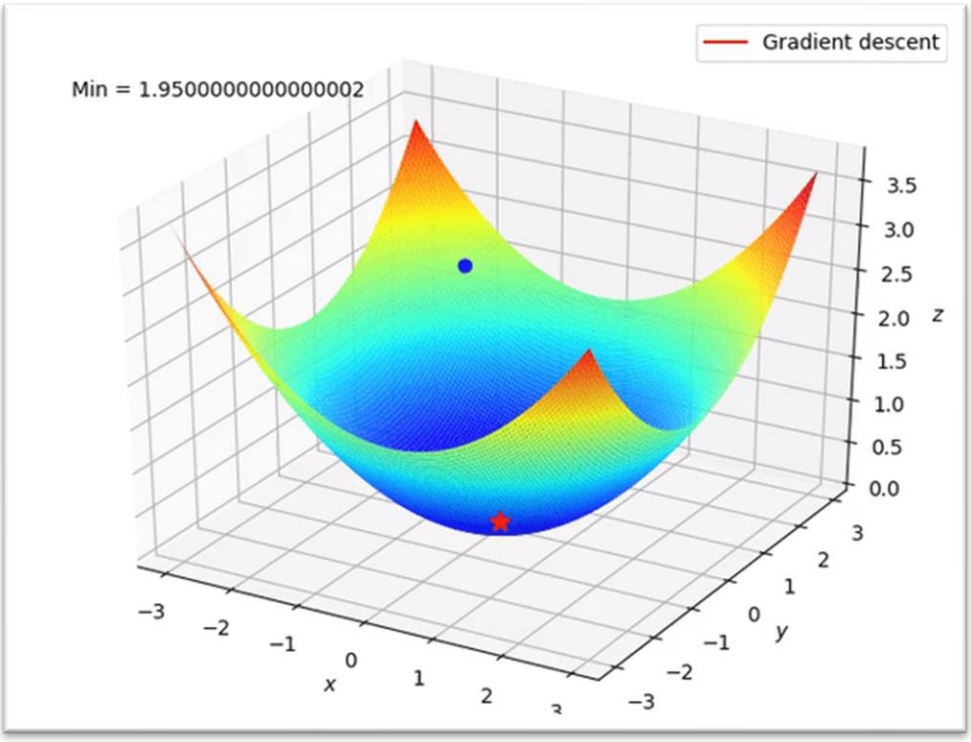
Conversely, with x_old = 4, f'(4) > 0 => x_new <x_old, so the solution will move to the left, closer to the minimum point.

- **Gradient for 1 variable function:**



Through the pictures above, we can see that Gradient descent depends on many factors: for example, choosing a different initial x point will affect the convergence process; or the learning rate is too large or too small, it also has an impact: if the learning rate is too small, the convergence speed is very slow, affecting the training process, and if the learning rate is too large, it will quickly reach the target after a few minutes. However, the algorithm does not converge and loops around the destination because the jump is too large.

- Gradient descent for multi-variable function:



Advantages of Gradient Descent in Machine Learning:

- **Versatility:** Gradient Descent is a versatile optimization algorithm that can be applied to a wide range of machine learning problems, including regression, classification, and neural network training.
- **Efficiency:** It is computationally efficient, especially in scenarios where the dataset is large, as it processes data in small batches or even individual data points (stochastic gradient descent).
- **Scalability:** Gradient Descent can scale to high-dimensional parameter spaces, making it suitable for complex models with a large number of parameters, such as deep neural networks.
- **Convergence:** When appropriately tuned, Gradient Descent generally converges to a minimum of the cost function, which is crucial for finding optimal model parameters.

- **Global Optima:** In convex optimization problems, where the cost function has a single minimum, Gradient Descent is guaranteed to converge to the global minimum.

Disadvantages of Gradient Descent in Machine Learning:

- **Sensitivity to Learning Rate:** The choice of the learning rate is critical, and picking an inappropriate value can lead to slow convergence or divergence. Finding the right learning rate can sometimes be challenging.
- **Local Minima:** In non-convex optimization problems, Gradient Descent may converge to a local minimum rather than the global minimum, impacting the model's performance.
- **Initial Conditions:** The convergence of Gradient Descent can be influenced by the initial values of the parameters. Starting from poor initial conditions might result in suboptimal solutions.
- **Computational Intensity for Large Datasets:** Processing large datasets can be computationally intensive, especially when using the batch gradient descent approach, where the entire dataset is used in each iteration.
- **Non-Differentiable Functions:** Gradient Descent relies on the gradient of the cost function, making it unsuitable for problems where the cost function is not differentiable.
- **Vanishing or Exploding Gradients:** In deep neural networks, the gradients can become extremely small (vanishing gradient) or large (exploding gradient), causing issues during training.

Example with python:

Load libraries:

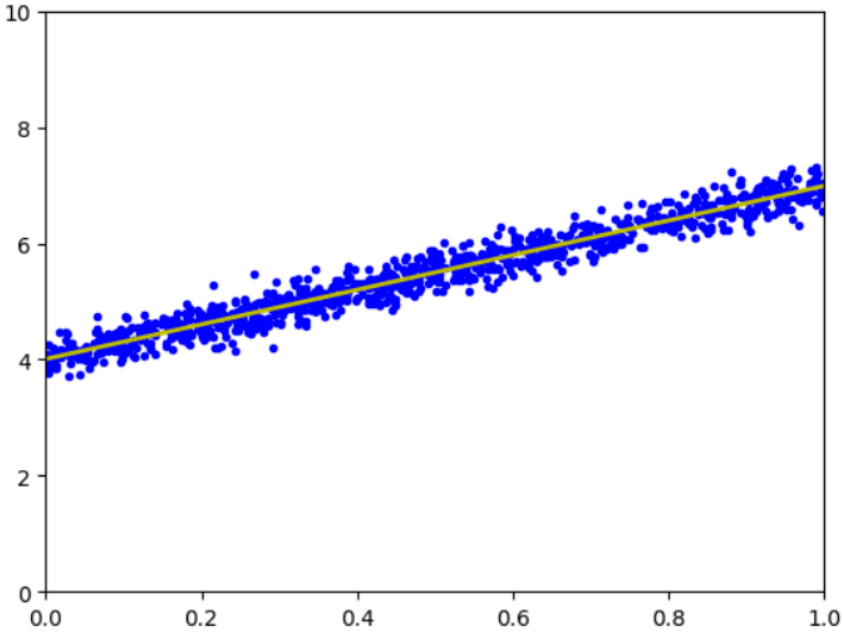
```
[2]  from __future__ import division, print_function, unicode_literals
     import numpy as np
     import matplotlib
     import matplotlib.pyplot as plt
     np.random.seed(2)
```

Next, we create 1000 selected data points close to the line y= 4 + 3x , display them and find the solution according to the formula:

```
X = np.random.rand(1000, 1)
y = 4 + 3 * X + .2*np.random.randn(1000, 1) # noise added

# Building Xbar
one = np.ones((X.shape[0],1))
Xbar = np.concatenate((one, X), axis = 1)

A = np.dot(Xbar.T, Xbar)
b = np.dot(Xbar.T, y)
w_lr = np.dot(np.linalg.pinv(A), b)
print('Solution found by formula: w = ',w_lr.T)

# Display result
w = w_lr
w_0 = w[0][0]
w_1 = w[1][0]
x0 = np.linspace(0, 1, 2, endpoint=True)
y0 = w_0 + w_1*x0

# Draw the fitting line
plt.plot(X.T, y.T, 'b.')     # data
plt.plot(x0, y0, 'y', linewidth = 2)   # the fitting line
plt.axis([0, 1, 0, 10])
plt.show()
```

Out results:

```
Solution found by formula: w =  [[4.0071715  2.98225924]]
```



The straight line found is the yellow line with the equation $y \approx 4 + 2.998x$.

Next we write the derivative and loss function:

```python
def grad(w):
    N = Xbar.shape[0]
    return 1/N * Xbar.T.dot(Xbar.dot(w) - y)

def cost(w):
    N = Xbar.shape[0]
    return .5/N*np.linalg.norm(y - Xbar.dot(w), 2)**2;
```

The code is simple to check the derivative and can be applied to any function (of a vector) with the cost and grad calculated above.

```python
def numerical_grad(w, cost):
    eps = 1e-4
    g = np.zeros_like(w)
    for i in range(len(w)):
        w_p = w.copy()
        w_n = w.copy()
        w_p[i] += eps
        w_n[i] -= eps
        g[i] = (cost(w_p) - cost(w_n))/(2*eps)
    return g

def check_grad(w, cost, grad):
    w = np.random.rand(w.shape[0], w.shape[1])
    grad1 = grad(w)
    grad2 = numerical_grad(w, cost)
    return True if np.linalg.norm(grad1 - grad2) < 1e-6 else False

print( 'Checking gradient...', check_grad(np.random.rand(2, 1), cost, grad))
```

```
Checking gradient... True
```

1.4.RMSporop

RMSprop (Root Mean Square Propagation) is an optimization method used in the process of training machine learning models, especially in models that use gradient descent. RMSprop is designed to reduce the problem of Adagrad, another optimization method, which involves adjusting the learning rate of parameters according to their frequency of occurrence.

RMSprop solves the above problem by using a moving average of squared gradients to normalize the gradient. This normalization balances the step size (momentum), decreasing the step for large gradients to avoid exploding and increasing the step for small gradients to avoid fading.

How RMSprop works:

In machine learning, when we train a model, we calculate gradients to understand the direction and steepness of the slope (error) for each parameter. These gradients tell us how much we should adjust the parameters to improve the model's performance.

In RMSprop, firstly, we square each gradient, which helps us focus on the positive values and removes any negative signs. We then calculate the average of all the squared gradients over some recent steps. This average tells us how fast the gradients have been changing and helps us understand the overall behaviour of the slopes over time.

Now, instead of using a fixed learning rate for all parameters, RMSprop adjusts the learning rate for each parameter separately. It does this by taking the average of squared gradients we calculated earlier and using it to divide the learning rate. This division makes the learning rate bigger when the average squared gradient is smaller and smaller when the average squared gradient is bigger.

Mathematical expression:

For each parameter in the model, let us calculate the squared gradient at time step t:

g^2 = gradient of the parameter at time step t

Then, let us calculate the exponential moving average of squared gradients:

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

Here, β is a hyperparameter between 0 and 1. It controls how much historical information to retain. Generally, it is set to a value like 0.9.

Now, for each parameter in the model, let us update it using the following formula:

$parameter_t = parameter_{\{t-1\}} - (learning\ rate / sqrt(E[g^2]_t + \varepsilon)) * g_t$

Here, $parameter_t$ represents the value of the parameter at time step t, and ϵ is a small constant (usually around 10^{-8}) added to the denominator to prevent division by zero.

Let's look at some of the above-mentioned algorithms and see why RMSprop is a preferred choice for optimizing neural networks and ML models.

Comparison with Gradient Descent:

Continuing with the analogy of navigating a valley, imagine taking substantial steps in random directions as you cannot perceive the exact location of the valley. Over time, you observe that certain directions exhibit steeper slopes, while others are more gradual. To optimize your journey, you start adapting the size of your steps based on the steepness of the slope. In regions with steep slopes, you take smaller steps to prevent overshooting, whereas in areas with gentler slopes, you take larger steps. This iterative adjustment, known as gradient descent, guides you toward the local minimum of a differentiable function by moving in the direction of the steepest descent.

Now, let's introduce RMSprop into the analogy. As you proceed, you maintain a record of the historical slopes encountered in each direction. Rather than solely adjusting step size based on the current slope, you consider how slopes have evolved over time.

Imagine a small ball rolling down the valley. When the ball traverses steep slopes, it gains speed, and on flatter slopes, it decelerates. By gauging the ball's speed, you can deduce the steepness of the valley. In the context of RMSprop, the ball symbolizes the history of gradients or slopes in each direction.

RMSprop retains an estimation of the average of squared gradients for each parameter.

Continuing the journey, this information becomes instrumental in determining step sizes in each direction. If the average squared gradient is substantial, signifying that the ball is rolling swiftly and encountering steep slopes, smaller steps are taken to prevent overshooting. Conversely, if the average squared gradient is small, indicating that the ball is moving slowly on gentler slopes, larger steps can be taken.

Through this adaptive adjustment of step sizes, RMSprop efficiently guides the exploration towards the valley's bottom, enhancing the effectiveness of the optimization process.

Advantages of RMSprop Optimizer:

- **Adaptive Learning Rate:** RMSprop automatically adapts the learning rate for each parameter based on the historical information of gradients. This adaptive behavior helps in handling different scales of gradients and speeds up convergence.
- **Mitigation of Vanishing/Exploding Gradients:** By maintaining an exponentially decaying average of squared gradients, RMSprop mitigates issues related to vanishing or exploding gradients in deep neural networks, making it suitable for training deep learning models.
- **Stability in Learning Process:** RMSprop adds a level of stability to the learning process by taking into account the historical information of gradients. This helps prevent drastic changes in the learning rate and provides smoother convergence.
- **Effective on Non-Convex Surfaces:** RMSprop is effective in navigating non-convex optimization surfaces, where the objective function may

have various local minima, by dynamically adjusting the learning rates based on the curvature of the loss landscape.

Advantages of RMSprop Optimizer:

- Accumulation of squared gradients: RMSprop keeps track of squared gradients over time, which can lead to very small learning rates during later stages of training, slowing down the optimization process.
- Hyperparameter sensitivity: RMSprop depends on a parameter (β) that can be difficult to set properly, affecting the algorithm's performance.
- Lack of momentum: Unlike some other optimization methods, RMSprop lacks momentum, which can help the learning process by accumulating past gradients.
- Limited adaptation to parameters: RMSprop treats all parameters equally when adapting learning rates, which may not be ideal in some situations.
- Sensitive to learning rate: RMSprop is sensitive to the initial learning rate setting, and choosing the wrong value can cause problems.
- Potential for getting stuck: Like many optimization methods, RMSprop can get stuck in local minima, hindering finding the best solution.

Example code RMSprop optimizer:

```
# Generate some dummy data
import numpy as np
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Build a simple linear regression model
model = Sequential()
model.add(Dense(units=1, input_dim=1))

# Compile the model with RMSprop optimizer
optimizer = RMSprop(learning_rate=0.001, rho=0.9)
model.compile(optimizer=optimizer, loss='mean_squared_error')

# Train the model
model.fit(X, y, epochs=50, batch_size=10)

# Make predictions
predictions = model.predict(X)

# Print the learned parameters
print("Learned slope (m):", model.layers[0].get_weights()[0][0, 0])
print("Learned intercept (b):", model.layers[0].get_weights()[1][0])
```

Out results:

```
Epoch 1/50
10/10 [==============================] - 1s 7ms/step - loss: 69.6964
Epoch 2/50
10/10 [==============================] - 0s 4ms/step - loss: 69.2564
Epoch 3/50
10/10 [==============================] - 0s 3ms/step - loss: 68.9089
Epoch 4/50
10/10 [==============================] - 0s 4ms/step - loss: 68.5793
Epoch 5/50
10/10 [==============================] - 0s 4ms/step - loss: 68.2481
Epoch 6/50
10/10 [==============================] - 0s 2ms/step - loss: 67.9193
Epoch 7/50
10/10 [==============================] - 0s 2ms/step - loss: 67.5972
Epoch 8/50
10/10 [==============================] - 0s 2ms/step - loss: 67.2753
Epoch 9/50
10/10 [==============================] - 0s 2ms/step - loss: 66.9493
Epoch 10/50
10/10 [==============================] - 0s 2ms/step - loss: 66.6274
Epoch 11/50
10/10 [==============================] - 0s 2ms/step - loss: 66.3052
Epoch 12/50
10/10 [==============================] - 0s 2ms/step - loss: 65.9859
Epoch 13/50
10/10 [==============================] - 0s 2ms/step - loss: 65.6689
Epoch 14/50

Epoch 17/50
10/10 [==============================] - 0s 3ms/step - loss: 64.3992
Epoch 18/50
10/10 [==============================] - 0s 3ms/step - loss: 64.0808
Epoch 19/50
10/10 [==============================] - 0s 2ms/step - loss: 63.7635
Epoch 20/50
10/10 [==============================] - 0s 2ms/step - loss: 63.4507
Epoch 21/50
10/10 [==============================] - 0s 2ms/step - loss: 63.1383
Epoch 22/50
10/10 [==============================] - 0s 2ms/step - loss: 62.8232
Epoch 23/50
10/10 [==============================] - 0s 2ms/step - loss: 62.5091
Epoch 24/50
10/10 [==============================] - 0s 2ms/step - loss: 62.1935
Epoch 25/50
10/10 [==============================] - 0s 2ms/step - loss: 61.8777
Epoch 26/50
10/10 [==============================] - 0s 2ms/step - loss: 61.5671
Epoch 27/50
10/10 [==============================] - 0s 2ms/step - loss: 61.2652

10/10 [==============================] - 0s 2ms/step - loss: 54.3637
4/4 [==============================] - 0s 4ms/step
Learned slope (m): -0.73571867
Learned intercept (b): 0.5056134
```

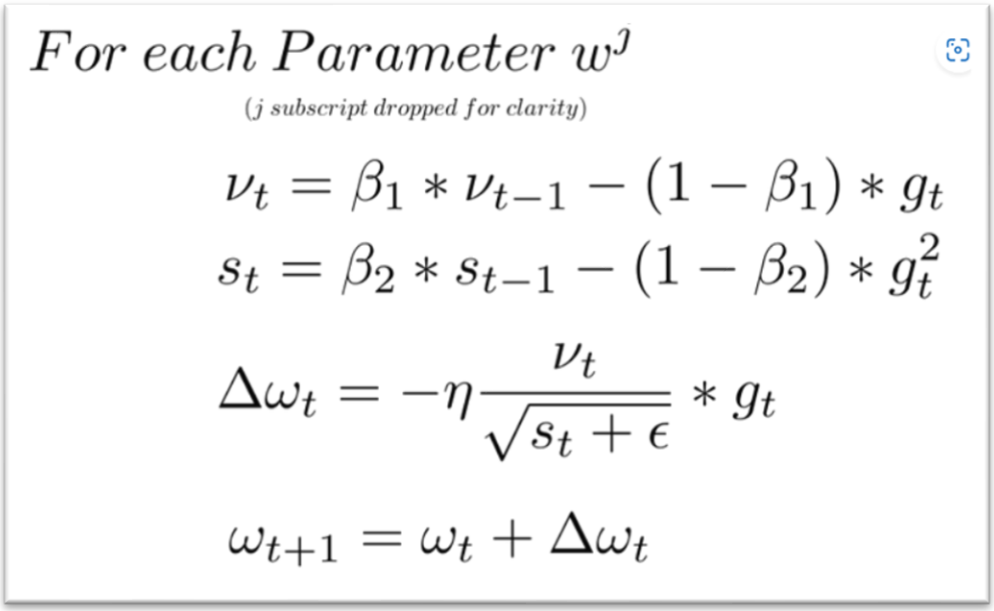
1.5. Adam

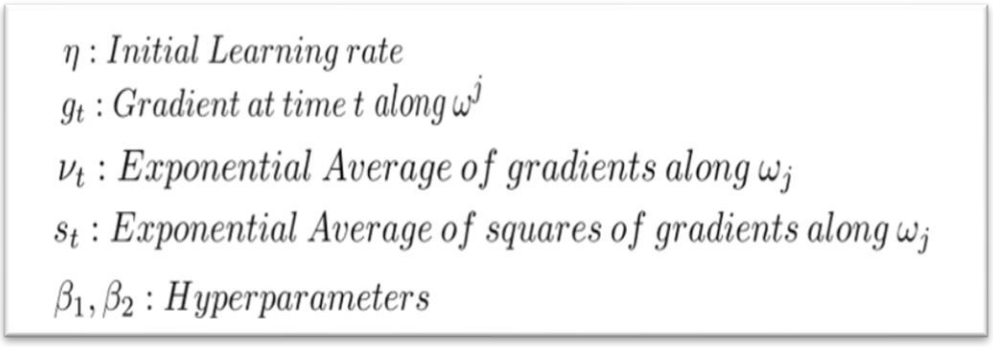
The Adam optimizer is a popular algorithm used in deep learning that helps adjust the parameters of a neural network in real-time to improve its accuracy and speed. Adam stands for Adaptive Moment Estimation, which means that it adapts the learning rate of each parameter based on its historical gradients and momentum.

In simple terms, Adam uses a combination of adaptive learning rates and momentum to make adjustments to the network's parameters during training. This helps the neural network learn faster and converge more quickly towards the optimal set of parameters that minimize the cost or loss function.

Adam is known for its fast convergence and ability to work well on noisy and sparse datasets. It can also handle problems where the optimal solution lies in a wide range of parameter values.

Overall, the Adam optimizer is a powerful tool for improving the accuracy and speed of deep learning models. By analyzing the historical gradients and adjusting the learning rate for each parameter in real-time, Adam can help the neural network converge faster and more accurately during training.





Adam is effective:

One of the main advantages of Adam is its ability to handle noisy and sparse datasets, which are common in real-world applications. It can also handle problems where the optimal solution lies in a wide range of parameter values.

Studies have shown that Adam can often outperform other optimization methods, such as stochastic gradient descent and its variants, in terms of convergence speed and generalization to new data. However, the optimal choice of optimizer may depend on the specific dataset and problem being solved.

Overall, the Adam optimizer is a powerful tool for improving the accuracy and speed of deep learning models. Its adaptive learning rate and momentum-based approach can help the neural network learn faster and converge more quickly towards the optimal set of parameters that minimize the cost or loss function.

Adam Configuration Parameters:

The Adam optimizer has several configuration parameters that can be adjusted to improve the performance of a deep learning model. Here are some of the main Adam configuration parameters explained:

- Learning rate: This parameter controls how much the model's parameters are updated during each training step. A high learning rate can result in large updates to the model's parameters, which can cause the optimization process to become unstable. On the other hand, a low learning rate can result in slow convergence and may require more training steps to reach the optimal set of parameters.
- Beta1 and Beta2: These parameters control the exponential decay rates for the first and second moment estimates of the gradient, respectively. In other words, they help the optimizer keep track of the historical gradients for each parameter during training.
- Epsilon: This parameter is used to avoid dividing by zero when calculating the update rule. It is a small value added to the denominator of the update rule to ensure numerical stability.
- Weight decay: This is a regularization term that can be added to the cost function during training to prevent overfitting. It penalizes large values of the model's parameters, which can help improve generalization to new data.

- Batch size: This parameter controls how many training examples are used in each training step. A larger batch size can result in faster convergence, but can also increase memory requirements and slow down the training process.
- Max epochs: This parameter determines the maximum number of training epochs or iterations that will be performed during training. It helps prevent overfitting and can improve the generalization of the model to new data.

The Adam Algorithm for Stochastic Optimization

The Adam optimizer, employed in stochastic optimization for deep learning, aims to enhance the effectiveness of gradient-based optimization methods like stochastic gradient descent. This algorithm operates by managing two dynamic averages: one for the gradient and another for the square of the gradient. These moving averages serve to estimate the first and second moments of the gradient, providing insights into its behavior.

In the course of each training iteration, the algorithm computes the gradient of the cost or loss function concerning each model parameter. Subsequently, it updates the moving averages and determines an adaptive learning rate for each parameter. This adaptive learning rate, informed by the historical information encoded in the moving averages, contributes to more efficient and nuanced weight updates during the training process.

Visual Comparison Between Optimizers

Several widely used optimization techniques in machine learning include stochastic gradient descent (SGD), momentum, Nesterov accelerated gradient (NAG), AdaGrad, AdaDelta, and RMSprop.

SGD, a straightforward optimization method, updates model parameters based on the cost function's gradient with respect to the parameters. Despite its popularity due to simplicity, SGD may exhibit slow convergence and susceptibility to local minima entrapment.

Momentum optimization, a variant of SGD, introduces a momentum term to parameter updates, aiding the optimizer in accelerating towards the minimum and overcoming small gradients or data noise. However, it can encounter challenges such as getting stuck in saddle points or oscillating around the minimum.

NAG, another momentum optimization variant, employs a "look-ahead" strategy to estimate the cost function's gradient at the next step. This approach helps prevent overshooting the minimum, potentially leading to faster convergence, although it may suffer from oscillation or instability.

AdaGrad adapts the learning rate for each parameter based on historical gradients, facilitating smaller updates for frequently adjusted parameters and larger updates for less frequently updated ones. Nevertheless, AdaGrad may exhibit a slow convergence rate and an excessive decrease in the learning rate over time.

AdaDelta, a variant of AdaGrad, introduces an exponential decay rate to control the adaptive learning rate, addressing AdaGrad's decreasing learning rate problem and potentially leading to faster convergence. However, it may be slower than other optimizers in the early stages of training.

RMSprop utilizes a moving average of squared gradients to adapt the learning rate for each parameter. This enables overcoming AdaGrad's slow convergence and handling non-stationary problems. Nonetheless, RMSprop can pose challenges such as high memory requirements and instability in certain cases.

Advantages of Adam Optimizer:

- **Adaptive Learning Rates:** Adam dynamically adjusts the learning rates for each parameter based on the past gradients and squared gradients. This adaptability allows for faster convergence and efficient optimization, especially in scenarios with varying gradients.
- **Combination of Momentum and RMSprop:** Adam combines the benefits of momentum optimization and RMSprop, incorporating momentum-like behavior to handle sparse gradients and using the moving average of squared gradients for adaptive learning rates. This makes it well-suited for a wide range of optimization problems.
- **Effective on Noisy or Sparse Data:** Adam's adaptive learning rate mechanism makes it effective in scenarios with noisy or sparse gradients, where traditional optimization methods may struggle.
- **Robust Performance:** Adam often exhibits robust performance across different types of neural network architectures and tasks. It is widely used in practice and is considered a reliable optimizer.
- **Suitable Default Parameters:** Adam's default hyperparameters often work well across various tasks, reducing the need for extensive hyperparameter tuning.

Disadvantages of Adam Optimizer:

- **Sensitivity to Learning Rate:** Adam can be sensitive to the choice of the initial learning rate. In some cases, a learning rate that is too high may lead to oscillations or divergence, while a rate that is too low may result in slow convergence.
- **Memory Intensive:** Adam maintains additional moving averages for each parameter, leading to higher memory requirements compared to

simpler optimizers like SGD. This can be a limitation in memory-constrained environments, especially for large models.

- **Not Always the Fastest Convergence:** While Adam often converges quickly, it may not always converge faster than other optimizers, especially in well-behaved optimization landscapes. In some cases, simpler optimizers like SGD with momentum or RMSprop may perform similarly or even outperform Adam.
- **Not Ideal for All Problems:** There are instances where Adam may not be the ideal choice. For example, in certain non-convex optimization scenarios, Adam might struggle to escape saddle points.
- **Hyperparameter Sensitivity:** While Adam has default parameters that work well in many cases, its performance can be sensitive to the choice of hyperparameters, and finding the right set may require experimentation.

Example adam optimizer with python:

```python
import torch
from torch import nn
import torch.optim as optim
a = 2.4785694
b = 7.3256989
error = 0.1
n = 100

# Data
x = torch.randn(n, 1)

t = a * x + b + (torch.randn(n, 1) * error)

model = nn.Linear(1, 1)
optimizer = optim.Adam(model.parameters(), lr=0.05)
loss_fn = nn.MSELoss()

# Run training
niter = 10
for _ in range(0, niter):
  optimizer.zero_grad()
  predictions = model(x)
  loss = loss_fn(predictions, t)
  loss.backward()
  optimizer.step()

  print("-" * 10)
  print("learned a = {}".format(list(model.parameters())[0].data[0, 0]))
  print("learned b = {}".format(list(model.parameters())[1].data[0]))
```

Out results:

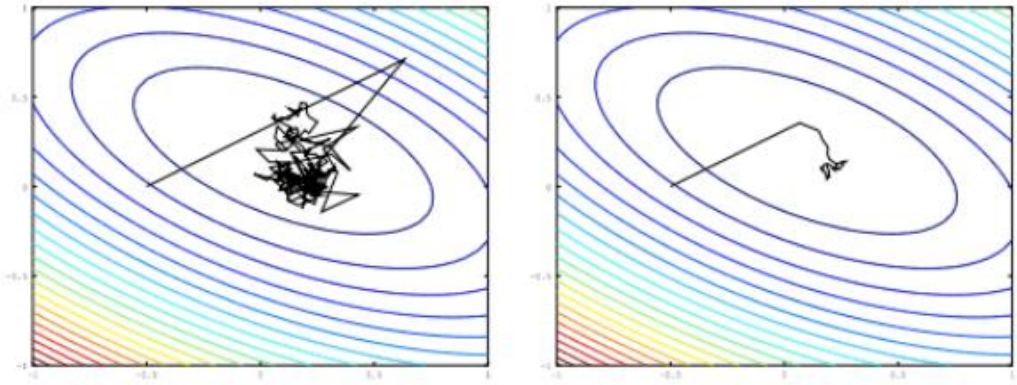
```
----------
learned a = -0.7452252507209778
learned b = -0.0521901398897171
----------
learned a = -0.6952490210533142
learned b = -0.002198968082666397
----------
learned a = -0.645313024520874
learned b = 0.04777742922306061
----------
learned a = -0.5954338908195496
learned b = 0.09773308783769608
----------
learned a = -0.5456286668777466
learned b = 0.14766202867031097
----------
learned a = -0.49591463804244995
learned b = 0.19755825400352478
----------
learned a = -0.44630929827690125
learned b = 0.24741582572460175
----------
learned a = -0.39683035016059875
learned b = 0.29722878336906433
----------
learned a = -0.3474956452846527
learned b = 0.34699124097824097
----------
learned a = -0.2983231544494629
learned b = 0.39669737219810486
```

1.6. Adagrad.

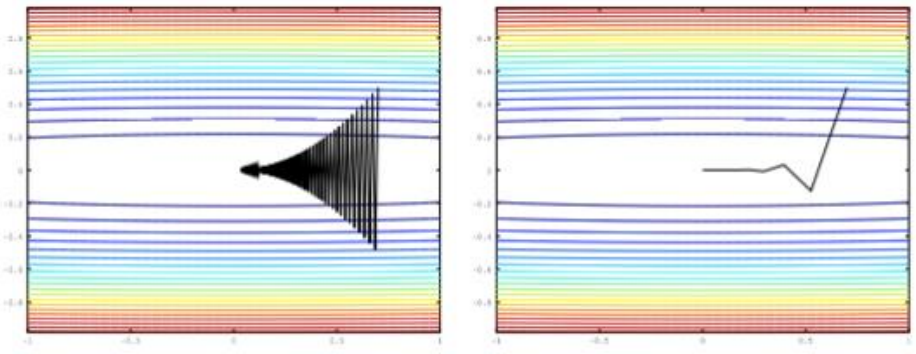
AdaGrad is simply just an optimization method based off of the Proximal Point Algorithm (otherwise known as the Gradient Descent algorithm), specifically the Stochastic version of gradient descent. The intention behind the formulation of AdaGrad is because SGD (stochastic gradient descent) converges slowly in the cases when features of our model are significantly more important than others. This is because the original SGD algorithm treats all factors equally,

regardless of how much each factor contributes towards the actual models outputs.

Seen below, we can recognize that the stochastic gradient descent algorithm takes a large amount of time in order to find the optimal solution because of the constant jumping back and forth on the expected gradient (seen Left). On the otherhand, if we were to average all of the iterates in SGD (not AdaGrad), we can find a much quicker optimization and decrease the amount of backtracking done.



Now consider a case where we have an extremely wide problem domain, most likely due to a feature having an exaggerated effect on the domain. Seen below, we can see that the standard proximal gradient method (SGD) ends up jumping back and forth because it is a poorly conditioned problem. On the other hand, we can see the AdaGrad variant (which somewhat fixes the poor conditioning of the problem) will end up getting to the optimal in a significantly smaller number of steps.



The original SGD algorithm's update method defines the update rule as:

$$x^{(k+1)} = \text{argmin}_{x \in \chi}\{\langle g^{(k)}, x\rangle + \frac{1}{2\alpha_k}||x - x^{(k)}||_2^2\}$$

This version of the algorithm has a general convergence rate of $O(1/\sqrt{T})$

AdaGrad takes advantage of the matrix norm, in our case, B will be a positive definite matrix such that:

$$||x||_B^2 := x^T Bx \geq 0$$

This matrix B has a list of diagonal terms that will modify the update rule of SGD such that:

$$x^{(k+1)} = \text{argmin}_{x \in \chi}\{\langle \nabla f(x^{(k)}), x\rangle + \frac{1}{2\alpha_k}||x - x^{(k)}||_B^2\}$$

which is equivalent to an update rule of:

$$x^{(k+1)} = x^{(k)} - \alpha B^{-1}\nabla f(x^{(k)})$$

As can be seen when compared to the original update rule, we have introduced the diagonal matrix B in order to converge more quickly because of the reliance on features that are more exaggerated than less important features. Now the question is how we can best approximate B, since it is unknown to the user. AdaGrad now extends the SGD algorithm by defining within iteration k:

$$G^{(k)} = \text{diag}[\sum_{i=1}^{k} g^{(i)}(g^{(i)})^T]^{1/2}$$

We can define $G^{(k)}$ as a diagonal matrix with entries:

$$G_{jj}^{(k)} = \sqrt{\sum_{i=1}^{k}(g_j^{(i)})^2}$$

Therefore modifying our previous modified update rule to be our final update rule for Adagrad:

$$x^{(k+1)} = \text{argmin}_{x\in\chi}\{\langle\nabla f(x^{(k)}), x\rangle + \tfrac{1}{2\alpha_k}||x - x^{(k)}||^2_{G^{(k)}}\}$$

which is equivalent to an update rule of:

$$x^{(k+1)} = x^{(k)} - \alpha G^{-1}\nabla f(x^{(k)})$$

Advantages of Adagrad Optimizer:

- **Adaptive Learning Rates:** Adam dynamically adjusts the learning rates for each parameter based on the past gradients and squared gradients. This adaptability allows for faster convergence and efficient optimization, especially in scenarios with varying gradients.
- **Combination of Momentum and RMSprop:** Adam combines the benefits of momentum optimization and RMSprop, incorporating momentum-like behavior to handle sparse gradients and using the

moving average of squared gradients for adaptive learning rates. This makes it well-suited for a wide range of optimization problems.

- **Effective on Noisy or Sparse Data:** Adam's adaptive learning rate mechanism makes it effective in scenarios with noisy or sparse gradients, where traditional optimization methods may struggle.
- **Robust Performance:** Adam often exhibits robust performance across different types of neural network architectures and tasks. It is widely used in practice and is considered a reliable optimizer.
- **Suitable Default Parameters:** Adam's default hyperparameters often work well across various tasks, reducing the need for extensive hyperparameter tuning.

Disadvantages of Adagrad Optimizer:

- **Sensitivity to Learning Rate:** Adam can be sensitive to the choice of the initial learning rate. In some cases, a learning rate that is too high may lead to oscillations or divergence, while a rate that is too low may result in slow convergence.
- **Memory Intensive:** Adam maintains additional moving averages for each parameter, leading to higher memory requirements compared to simpler optimizers like SGD. This can be a limitation in memory-constrained environments, especially for large models.
- **Not Always the Fastest Convergence:** While Adam often converges quickly, it may not always converge faster than other optimizers, especially in well-behaved optimization landscapes. In some cases, simpler optimizers like SGD with momentum or RMSprop may perform similarly or even outperform Adam.
- **Not Ideal for All Problems:** There are instances where Adam may not be the ideal choice. For example, in certain non-convex optimization scenarios, Adam might struggle to escape saddle points.
- **Hyperparameter Sensitivity:** While Adam has default parameters that work well in many cases, its performance can be sensitive to the choice of hyperparameters, and finding the right set may require experimentation.

Example adagrad optimizer with python:

```python
import tensorflow as tf
from tensorflow.keras.optimizers import Adagrad
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Generate some dummy data
import numpy as np
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Build a simple linear regression model
model = Sequential()
model.add(Dense(units=1, input_dim=1))

# Compile the model with AdaGrad optimizer
optimizer = Adagrad(learning_rate=0.01, initial_accumulator_value=0.1, epsilon=1e-07)
model.compile(optimizer=optimizer, loss='mean_squared_error')

# Train the model
model.fit(X, y, epochs=50, batch_size=10)

# Make predictions
predictions = model.predict(X)

# Print the learned parameters
print("Learned slope (m):", model.layers[0].get_weights()[0][0, 0])
print("Learned intercept (b):", model.layers[0].get_weights()[1][0])
```

Our results:

```
Epoch 1/50
10/10 [==============================] - 1s 4ms/step - loss: 69.3188
Epoch 2/50
10/10 [==============================] - 0s 3ms/step - loss: 68.1748
Epoch 3/50
10/10 [==============================] - 0s 2ms/step - loss: 67.4429
Epoch 4/50
10/10 [==============================] - 0s 2ms/step - loss: 66.8545
Epoch 5/50
10/10 [==============================] - 0s 2ms/step - loss: 66.3487
Epoch 6/50
10/10 [==============================] - 0s 3ms/step - loss: 65.8988
Epoch 7/50
10/10 [==============================] - 0s 3ms/step - loss: 65.4897
Epoch 8/50
10/10 [==============================] - 0s 3ms/step - loss: 65.1131
Epoch 9/50
10/10 [==============================] - 0s 3ms/step - loss: 64.7622
Epoch 10/50
10/10 [==============================] - 0s 2ms/step - loss: 64.4328
Epoch 11/50
10/10 [==============================] - 0s 2ms/step - loss: 64.1210
Epoch 12/50
10/10 [==============================] - 0s 2ms/step - loss: 63.8245
Epoch 13/50
10/10 [==============================] - 0s 4ms/step - loss: 63.5417
```

```
Epoch 25/50
10/10 [==============================] - 0s 3ms/step - loss: 60.8125
Epoch 26/50
10/10 [==============================] - 0s 4ms/step - loss: 60.6228
Epoch 27/50
10/10 [==============================] - 0s 3ms/step - loss: 60.4366
Epoch 28/50
10/10 [==============================] - 0s 2ms/step - loss: 60.2547
Epoch 29/50
10/10 [==============================] - 0s 3ms/step - loss: 60.0761
Epoch 30/50
10/10 [==============================] - 0s 3ms/step - loss: 59.9011
Epoch 31/50
10/10 [==============================] - 0s 3ms/step - loss: 59.7298
Epoch 32/50
10/10 [==============================] - 0s 3ms/step - loss: 59.5610
Epoch 33/50
10/10 [==============================] - 0s 2ms/step - loss: 59.3957
```

```
Epoch 42/50
10/10 [==============================] - 0s 3ms/step - loss: 58.0196
Epoch 43/50
10/10 [==============================] - 0s 3ms/step - loss: 57.8777
Epoch 44/50
10/10 [==============================] - 0s 3ms/step - loss: 57.7377
Epoch 45/50
10/10 [==============================] - 0s 3ms/step - loss: 57.5996
Epoch 46/50
10/10 [==============================] - 0s 3ms/step - loss: 57.4631
Epoch 47/50
10/10 [==============================] - 0s 3ms/step - loss: 57.3282
Epoch 48/50
10/10 [==============================] - 0s 3ms/step - loss: 57.1951
Epoch 49/50
10/10 [==============================] - 0s 3ms/step - loss: 57.0636
Epoch 50/50
10/10 [==============================] - 0s 3ms/step - loss: 56.9338
4/4 [==============================] - 0s 3ms/step
Learned slope (m): -0.8296323
Learned intercept (b): 0.422787
```

CHAPTER 2 – CONTINUAL LEARNING AND TEST PRODUCTION.

2.1. Continual Learning

2.1.1. What is Continual Learning?

Continuous learning represents a contemporary paradigm within the field of machine learning, aiming to develop models capable of ongoing evolution and adaptation. In contrast to traditional machines that acquire fixed strategies, continuous learning enables models to adapt and accumulate knowledge over time, incorporating new insights without discarding past experiences. This mirrors the human learning process, where individuals continuously build upon their existing knowledge base.

The primary challenge addressed by continuous learning is the issue of severe forgetting, a phenomenon observed in conventional models that tend to lose proficiency in previously learned tasks when confronted with new missions. Continuous learning seeks to minimize this challenge, facilitating models in maintaining mastery over a spectrum of tasks and ensuring their relevance and effectiveness in an ever-changing environment. This approach empowers AI systems to stay applicable and resilient in dynamic and evolving scenarios.

The practical programs of chronic mastering are diverse and ways-accomplishing. In the realm of herbal language information, it permits chatbots and language models to maintain up with evolving linguistic developments and person interactions, ensuring greater correct and contextually relevant responses. In imaginative and prescient view, it allows recognition systems to adapt to new gadgets, environments, and visible standards, making them extra sturdy and versatile. Furthermore, within the area of independent robotics, persistent mastering equips machines with the functionality to examine from stories and adapt to distinctive obligations and environments, making them greater self-reliant and flexible in real-international applications. In essence, chronic studying is a fundamental step towards developing clever structures that could thrive in our ever-evolving, dynamic international.

2.1.2. Key factors in chronic gaining knowledge in system mastering include:

- Elastic Weight Consolidation (EWC): EWC retains the model's important weights for the previous task by imposing constraints on them during learning of the new task.

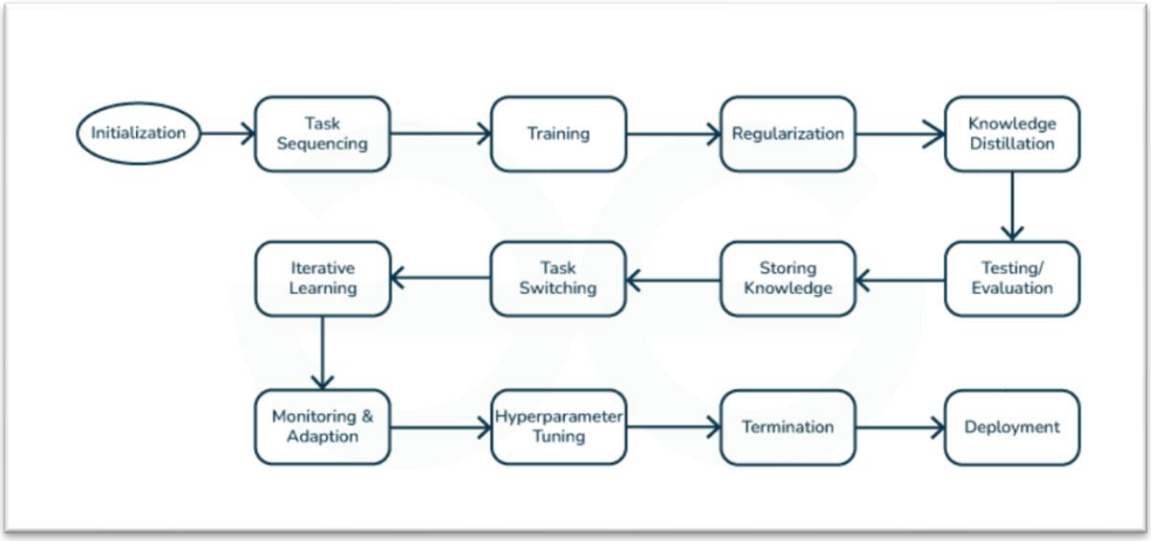
- Learning Without Forgetting (LwF): LwF retains the model's knowledge for the previous task by combining new and old data during the learning process.
- Gradient Episodic Memory (GEM): GEM retains important gradients of past models to avoid forgetting when learning new tasks.
- Replay-based Methods: Reuse data from previous tasks to help the model maintain knowledge.
- Dynamic Architectures: Expands the model's architecture when there are new tasks to contain more information.

2.1.3. Types of Continual Learning.

- **Task-based Continual Learning:** In this method, a version learns a sequence of distinct obligations through the years. The model’s goal is to conform to each new undertaking while preserving knowledge of previously found out obligations. Techniques which includes Elastic Weight Consolidation (EWC) and Progressive Neural Networks (PNN) fall into this class.
- **Class-incremental Learning:** Class-incremental mastering specializes in managing new classes or classes of information over the years while keeping understanding of formerly seen lessons. This is common in packages like image recognition, in which new object training are brought periodically. Methods like iCaRL (Incremental Classifier and Representation Learning) are used for class-incremental mastering.
- **Domain-incremental Learning:** Domain-incremental gaining knowledge of deals with adapting to new records distributions or domain names. For example, in self sufficient robotics, a robotic may want to adapt

to different environments. Techniques for area variation and area-incremental learning are used to handle this state of affairs.

2.1.4. **Process of Continual Learning**



- **Initialization:** Begin with an preliminary version, often pretrained on a huge dataset to provide foundational understanding. This pretrained version serves as a place to begin for persistent studying.
- **Task Sequencing:** Define the series of responsibilities or information streams that the model will encounter. Each undertaking can constitute a distinct trouble, a new set of statistics, or a unique aspect of the general problem.
- **Training on a Task:** Train the model on the first task inside the series. This entails updating the version's parameters the usage of information precise to the current undertaking. Typically, popular education techniques, like gradient descent, are used.

- **Regularization for Knowledge Preservation**: To prevent catastrophic forgetting, follow regularization strategies. These may additionally consist of strategies like Elastic Weight Consolidation (EWC) or Synaptic Intelligence (SI) to defend important parameters related to beyond obligations.
- **Knowledge Distillation**: For magnificence-incremental or area-incremental getting to know, understanding distillation may be used to transfer information from the authentic version or instructor model to the current version, enabling it to inherit the know-how of formerly seen lessons or domain names.
- **Testing and Evaluation:** After training on a project, compare the model's performance at the present day mission to ensure it has found out correctly. This can also involve wellknown evaluation metrics applicable to the unique mission.
- **Storing Knowledge:** Depending on the approach chosen, you may shop facts or representations from beyond duties in outside reminiscence or buffers. This saved knowledge may be replayed or used to mitigate forgetting whilst gaining knowledge of new tasks.
- **Task Switching:** Move to the next undertaking within the predefined series and repeat steps 3 to 7. The model ought to adapt to the new venture at the same time as ensuring that its overall performance on previous responsibilities isn't always notably degraded.
- **Iterative Learning:** Continue this method iteratively for each mission within the series, keeping a balance among adapting to new records and preserving vintage expertise.
- **Monitoring and Adaptation**: Continuously display the model's overall performance and edition abilties. If the model indicates symptoms of

forgetting or negative performance on preceding obligations, remember adjusting the regularization, replay, or distillation techniques.

- **Hyperparameter Tuning:** Adjust hyperparameters as had to optimize the stability between adapting to new obligations and preserving vintage expertise. This might also involve satisfactory-tuning the getting to know charge, regularization strengths, and different parameters.
- **Termination or Expansion**: Determine the preventing situations for the continual gaining knowledge of procedure, that may consist of a hard and fast number of obligations or a dynamic method that permits for indefinite variation. Alternatively, enlarge the version's structure or ability to handle extra obligations if necessary.
- **Real-world Deployment:** Once the version has discovered from the whole sequence of responsibilities, it is able to be deployed in real-global programs, wherein it is able to adapt and hold mastering as new data and obligations are encountered.

2.1.5. Implementing Continual Learning in Machine Learning

Import libraries.

```
[3]  import numpy as np
     import tensorflow as tf
     from tensorflow.keras import layers
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.optimizers import Adam
     from tensorflow.keras.losses import SparseCategoricalCrossentropy
```

Our results:

```python
# Generate synthetic data for two tasks
def generate_data(task_id):
    np.random.seed(task_id)
    if task_id == 0:
        X = np.random.rand(100, 10)
        y = (X.sum(axis=1) > 5).astype(int)
    else:
        X = np.random.rand(100, 10)
        y = (X[:, :5].sum(axis=1) > 2.5).astype(int)
    return X, y

# Define the model architecture
def create_model():
    model = Sequential([
        layers.Dense(64, activation='relu', input_shape=(10,)),
        layers.Dense(2, activation='softmax')
    ])
    model.compile(optimizer=Adam(), loss=SparseCategoricalCrossentropy(), metrics=['accuracy'])
    return model

# Implement Elastic Weight Consolidation (EWC) for continual learning
class EWC:
    def __init__(self, model, tasks):
        self.model = model
        self.tasks = tasks
        self.ewc_lambda = 0.01  # EWC regularization hyperparameter
        self.previous_tasks_weights = []

        for layer in model.layers:
            self.previous_tasks_weights.append(tf.Variable(tf.zeros_like(layer.get_weights()[0]), trainable=False))
```

```python
    def calculate_fisher(self, task_data):
        # Calculate Fisher information for each weight in the model
        fisher = []
        for i, task in enumerate(self.tasks):
            X, y = task_data[i]
            fisher_task = []
            with tf.GradientTape(persistent=True) as tape:
                predictions = self.model(X)
                loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=predictions))
            for j, layer in enumerate(self.model.layers):
                gradients = tape.gradient(loss, layer.trainable_weights)
                fisher_task.append([tf.reduce_mean((grad ** 2)) for grad in gradients])
            fisher.append(fisher_task)
        return fisher

    def update_ewc_loss(self, task_data):
        fisher_information = self.calculate_fisher(task_data)
        losses = []
        for i, task in enumerate(self.tasks):
            for j, layer in enumerate(self.model.layers):
                # Calculate EWC loss for each weight
                losses.append(tf.reduce_sum(fisher_information[i][j] * tf.square(layer.trainable_weights[0] - self.previous_tasks_weights[
        return self.ewc_lambda * tf.add_n(losses)

    def update_previous_task_weights(self):
        for i, layer in enumerate(self.model.layers):
            self.previous_tasks_weights[i].assign(layer.get_weights()[0])
```

```
# Training loop for sequential learning
tasks = [generate_data(task_id) for task_id in range(2)]

model = create_model()
ewc = EWC(model, tasks)

for i, task in enumerate(tasks):
    X, y = task
    print(f"Training on task {i}")

    # Train on current task
    model.fit(X, y, epochs=10)

    # Update EWC loss and previous task weights
    ewc_loss = ewc.update_ewc_loss(tasks[:i + 1])
    model.add_loss(ewc_loss)
    ewc.update_previous_task_weights()

# Evaluate the final model on all tasks
for i, task in enumerate(tasks):
    X, y = task
    print(f"Evaluating on task {i}")
    accuracy = model.evaluate(X, y, verbose=0)[1]
    print(f"Accuracy: {accuracy}")
```

```
Training on task 0
Epoch 1/10
4/4 [==============================] - 1s 4ms/step - loss: 0.6952 - accuracy: 0.4400
Epoch 2/10
4/4 [==============================] - 0s 4ms/step - loss: 0.6886 - accuracy: 0.4800
Epoch 3/10
4/4 [==============================] - 0s 6ms/step - loss: 0.6842 - accuracy: 0.5000
Epoch 4/10
4/4 [==============================] - 0s 4ms/step - loss: 0.6797 - accuracy: 0.5300
Epoch 5/10
4/4 [==============================] - 0s 5ms/step - loss: 0.6760 - accuracy: 0.5400
Epoch 6/10
4/4 [==============================] - 0s 4ms/step - loss: 0.6721 - accuracy: 0.5600
Epoch 7/10
4/4 [==============================] - 0s 4ms/step - loss: 0.6679 - accuracy: 0.6100
Epoch 8/10
4/4 [==============================] - 0s 4ms/step - loss: 0.6647 - accuracy: 0.6600
Epoch 9/10
4/4 [==============================] - 0s 4ms/step - loss: 0.6631 - accuracy: 0.7200
Epoch 10/10
4/4 [==============================] - 0s 4ms/step - loss: 0.6620 - accuracy: 0.6700
```

2.1.6. Advantages and drawbacks of continual learning.

Advantages of Continual Learning

- **Adaptability**: Allows models to adapt and evolve over time to make them well-suited for applications in dynamic and changing environments. This adaptability is crucial in fields like autonomous robotics and natural language understanding.
- **Efficiency**: Instead of retraining models from scratch every time new data or tasks emerge it enables incremental updates which saves computational resources and time.
- **Knowledge Retention**: It mitigates the problem of catastrophic forgetting enabling models to retain knowledge of past tasks or experiences. This is valuable when dealing with long-term memory retention in AI systems.

- **Reduced Data Storage**: Techniques like generative replay reduces the need to store and manage large historical datasets making it more feasible to deploy continual learning in resource-constrained settings.
- **Versatility**: It is applied to a wide range of domains including natural language processing, computer vision, recommendation systems that makes it a versatile approach in AI.

Disadvantages of Continuous Learning:

- Big Challenges with Big Data: In problems with large amounts of data, maintaining knowledge can become difficult due to limited resources.
- High Complexity: Continuous Learning methods often require high complexity, especially when the model has to handle many different types of tasks.
- Problem with Very Different Tasks: In cases where the new task is too different, the model may have difficulty learning and retaining knowledge from previous tasks.

2.2.Test Production.

Test Production" in the context of building a machine learning solution often involves testing and ensuring the stability and performance of the model when deployed to a production environment. Below are some important steps and aspects Important things to consider when implementing the "Test Production" process in building a machine learning solution:

Prepare Production Data: Ensure that the data the model will encounter in the production environment is properly processed and prepared. Check the data for completeness, consistency, and representativeness.

Model Testing: Perform model testing with new data from the production environment. Ensure that the model is efficient and meets technical requirements.

Performance Testing: Evaluate the model's performance under real production conditions. Ensure that the model operates fast enough and accurately enough to meet demand.

Continuous Applicability Testing: Perform ongoing availability testing of the model. Ensure that the model can withstand new tasks and data without failure.

Error Control and Logging: Put error control and logging systems in place to track any issues when the model runs in a production environment. Provide warning or error messages for reporting and processed immediately.

Security Testing: Perform security testing to ensure that the model and data are not exposed to threats.Verify that security measures are working effectively.

Performance Testing: Evaluate the model's performance under heavy load and in real production conditions. Test the model's scalability when there are many users or high load.

Compatibility Testing: Ensure model compatibility with other systems in the production environment. Test compatibility with different software versions and infrastructure.

Set Control Points and Recovery: Build control points and recovery processes quickly if the model crashes. Test in Realistic Environments:

If possible, deploy the model in a real production environment to test every aspect of integration and deployment.

REFERENCE

1. Machine Learning in Production - Testing (applyingml.com)
2. Continual Learning in Machine Learning - GeeksforGeeks
3. Gradient Descent With AdaGrad From Scratch - MachineLearningMastery.com
4. Machine Learning cơ bản (machinelearningcoban.com)
5. OPTIMIZER METHODS IN TRAINING MACHINE LEARNING MODELS? - Search (bing.com)
6. Optimizer- Hiểu sâu về các thuật toán tối ưu (GD,SGD,Adam,..) (viblo.asia)
7. Machine Learning: tests and production - Gustavo Millen (millengustavo.github.io)