

koustav-B /  
Prodigal-AI

&lt;&gt; Code Issues Pull requests Actions Projects Wiki Security In

Prodigal-AI / Day16-30 / Resma\_Swain\_OT\_blog.md



koustav-B Add files via upload

51c38bb · 4 days ago



481 lines (335 loc) · 29.6 KB

Preview

Code

Blame

Raw



# Optimization Techniques: A Practical Guide

By, Resma Swain, Koustav Biswas on 24-03-2025

In this blog we'll explore three popular optimization algorithms—**Stochastic Gradient Descent (SGD)**, **Adam**, and **RMSProp**—and implement them from scratch. We'll also implement the optimizers using TensorFlow to see how they perform.

## 1. What Are Optimization Algorithms?

Optimization algorithms are methods used to minimize a loss function, which measures how well a model is performing. Think of the loss function as a landscape with hills and valleys. The goal is to find the lowest point in this landscape, which corresponds to the best set of model parameters. Optimization algorithms guide the model by determining the direction and size of the steps needed to reach this minimum.

In machine learning, these algorithms adjust the model's parameters (e.g., weights in a neural network) iteratively to reduce the loss. The choice of optimization algorithm can significantly impact the speed and quality of training.

## 2. Implementing Stochastic Gradient Descent (SGD)


### 2.1 What Is SGD?

Stochastic Gradient Descent (SGD) is one of the simplest and most widely used optimization algorithms. Unlike traditional gradient descent, which computes the gradient using the entire dataset, SGD uses a single data point (or a small batch) to estimate the gradient. This makes it computationally efficient, especially for large datasets.

## 2.2 How Does It Work?

SGD works by computing the gradient of the loss function for a single data point and updating the model parameters in the opposite direction of the gradient. The size of the step is controlled by the learning rate. If the learning rate is too large, the algorithm may overshoot the minimum; if it's too small, training may take too long.

## 2.3 SGD Implementation from scratch



```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0 # Normalize pixel values to
x_train = x_train.reshape(-1, 28 * 28) # Flatten the images
x_test = x_test.reshape(-1, 28 * 28)
y_train = to_categorical(y_train, 10) # One-hot encode labels
y_test = to_categorical(y_test, 10)

# Create the model
model = Sequential([
    Dense(128, activation='relu', input_shape=(28 * 28,)),
    Dense(10, activation='softmax')
])

# Define a custom optimizer from scratch
class CustomSGD:
    def __init__(self, learning_rate=0.01):
        self.learning_rate = learning_rate

    def update(self, weights, grads):
        """
        Update weights using gradient descent.
        :param weights: List of model weights (trainable variables).
        :param grads: List of gradients corresponding to the weights.
        """
```

```

    for i in range(len(weights)):
        weights[i].assign_sub(self.learning_rate * grads[i]) # w = w - lr

# Loss function (categorical cross-entropy)
def categorical_crossentropy(y_true, y_pred):
    return -tf.reduce_mean(tf.reduce_sum(y_true * tf.math.log(y_pred), axis=1))

# Training loop
def train_model(model, x_train, y_train, epochs=5, batch_size=32, learning_rate=0.01):
    optimizer = CustomSGD(learning_rate=learning_rate)
    num_samples = x_train.shape[0]
    num_batches = int(np.ceil(num_samples / batch_size))

    for epoch in range(epochs):
        print(f"Epoch {epoch + 1}/{epochs}")
        for batch in range(num_batches):
            # Get a batch of data
            start = batch * batch_size
            end = start + batch_size
            x_batch = x_train[start:end]
            y_batch = y_train[start:end]

            # Forward pass
            with tf.GradientTape() as tape:
                y_pred = model(x_batch, training=True)
                loss = categorical_crossentropy(y_batch, y_pred)

            # Backward pass (compute gradients)
            grads = tape.gradient(loss, model.trainable_variables)

            # Update weights using the custom optimizer
            optimizer.update(model.trainable_variables, grads)

            # Print loss every 100 batches
            if batch % 100 == 0:
                print(f"Batch {batch}/{num_batches}, Loss: {loss.numpy()}")

# Train the model
train_model(model, x_train, y_train, epochs=5, batch_size=32, learning_rate=0.01)

# Evaluate the model
def evaluate_model(model, x_test, y_test):
    y_pred = model(x_test)
    y_pred = tf.argmax(y_pred, axis=1)
    y_true = tf.argmax(y_test, axis=1)
    accuracy = tf.reduce_mean(tf.cast(tf.equal(y_pred, y_true), tf.float32))
    print(f"Test Accuracy: {accuracy.numpy()}")

```

```
evaluate_model(model, x_test, y_test)
```

## 2.4 Explanation

### Building a Simple Neural Network for MNIST Digit Classification

The code is all about building a simple neural network to classify handwritten digits from the MNIST dataset using TensorFlow. The MNIST dataset is a collection of 28x28 grayscale images of digits ranging from 0 to 9, along with their corresponding labels. The main goal here is to train a model that can accurately recognize and classify these digits.

#### 1. Importing Libraries and Loading Data

To get started, we import the necessary libraries. TensorFlow is the main library we use to build and train the neural network. We also use the `Sequential` model from Keras, which helps us create a straightforward stack of layers, and the `Dense` layer, which is also called "fully connected layer." The MNIST dataset is loaded using TensorFlow's built-in `mnist.load_data()` function, and we use the `to_categorical` function to convert the labels into one-hot encoded vectors. This means that instead of a label being a single number (like `2`), it becomes a list of 0s and 1s (like `[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`), where the position of the `1` indicates the digit.

#### 2. Preprocessing the Data

Next, we preprocess the data to make it ready for training. The pixel values of the images are normalized to a range of `[0, 1]` by dividing them by `255`. This step is important because it helps the model learn faster and more effectively. The 28x28 images are then flattened into 1D vectors of size 784 (since  $28 * 28 = 784$ ). This is necessary because the fully connected layers we're using need the input to be in a 1D format. The labels are also converted into one-hot encoded vectors, as mentioned earlier.

#### 3. Building the Neural Network

Now, we build the neural network model using the `Sequential` API. The model has two layers:

1. **First Layer:** A fully connected layer with 128 neurons and uses the ReLU activation function, which helps introduce non-linearity into the model.
2. **Second Layer:** The output layer, which has 10 neurons (one for each digit) and uses the softmax activation function. Softmax converts the output into a probability

distribution, so we can see how likely the the input image is each digit.

#### 4. Custom Optimizer for Stochastic Gradient Descent (SGD)

Instead of using TensorFlow's built-in optimizers, we define our own custom optimizer to perform Stochastic Gradient Descent (SGD). The optimizer is implemented as a class with an `update` method that adjusts the model's weights using the formula:

**`weights = weights - learning_rate * gradients`**

- `weights` : The current values of the model's parameters.
- `learning_rate` : A hyperparameter that controls the size of the update steps.
- `gradients` : The computed gradients of the loss with respect to the weights.

This is essentially how gradient descent works—it updates the weights in small steps to minimize the loss. The learning rate controls how big these steps are.

#### 5. Loss Function: Categorical Cross-Entropy

The loss function we use is called **categorical cross-entropy**. This measures how far off the model's predictions are from the true labels. In simpler terms, it tells us how well the model is doing at classifying the digits.

#### 6. Training the Model

The training process is done using a custom training loop. The dataset is split into batches, and for each batch:

1. The model makes predictions and calculates the loss.
2. The gradients of the loss with respect to the model's weights are computed using TensorFlow's `GradientTape`.
3. These gradients are then used to update the model's weights using our custom optimizer.

To keep track of how the training is going, the loss is printed every 100 batches.

#### 7. Evaluating the Model

Once the model is trained, its performance on the test set is evaluated. The model makes predictions for the test images, and we use `argmax` to convert these predictions into class labels (e.g., the digit with the highest probability). We then compare these predicted labels to the true labels and calculate the accuracy, which tells us what percentage of the test images the model classified correctly.

## 2.5 SGD Implementation using tensorflow



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical # Load and preprocess data

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.reshape(-1, 28 * 28)
x_test = x_test.reshape(-1, 28 * 28)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Create and compile the model
model = Sequential([Dense(128, activation='relu', input_shape=(28 * 28,)), Dense(10)])
model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])

# Train and evaluate the model
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.2)
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)
```

## 2.6 Explanation

The SGD optimizer in TensorFlow implements the **Stochastic Gradient Descent (SGD)** algorithm, which is the simplest optimization method. When you compile the model with `optimizer='sgd'`, TensorFlow uses a fixed learning rate to update the parameters based on the gradients of the loss function. SGD does not maintain any moving averages or adapt the learning rate for individual parameters, so it requires careful tuning of the learning rate to work well.

## 3. Implementing SGD with Momentum

### 3.1 What is SGD with Momentum?

Stochastic Gradient Descent (SGD) with Momentum is an optimization algorithm that enhances standard SGD by incorporating a velocity term. This momentum term helps accelerate convergence by allowing the optimizer to maintain some of its previous movement, thereby smoothing out updates and reducing oscillations in the optimization path.

## 3.2 How Does It Work?

SGD with Momentum maintains a moving average of past gradients to influence the current update:

1. Velocity Update: The velocity accumulates past gradients, controlled by a momentum factor.
2. Parameter Update: The velocity is then used to adjust the parameters in the optimization process.

This mechanism allows the optimizer to continue moving in directions of consistent descent, reducing fluctuations and improving convergence speed.

## 3.3 SGD with momentum Implementation from scratch

```
import numpy as np

class SGDWithMomentum:
    def __init__(self, learning_rate=0.01, momentum=0.9):
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.velocity = {}

    def update(self, params, grads):
        for key in params:
            if key not in self.velocity:
                self.velocity[key] = np.zeros_like(params[key])

            # Update velocity
            self.velocity[key] = self.momentum * self.velocity[key] - self.learning_rate * grads[key]

            # Update parameters
            params[key] += self.velocity[key]
```

## 3.4 Explanation

The `SGDWithMomentum` class is initialized with two main hyperparameters: `learning_rate`, which controls the step size for parameter updates, and `momentum`, which determines how much of the previous update is retained to smooth optimization.

Additionally, `self.velocity` is a dictionary used to store the momentum for each parameter.

The `update` method takes in two arguments: `params`, a dictionary of model parameters, and `grads`, a dictionary of gradients with respect to the loss function.

For each parameter, if the velocity is not initialized, it is set to zero. The velocity is then updated using the formula:


$$v_t = \beta v_{t-1} + (1 - \beta) \nabla L$$

where ( $v_t$ ) is the velocity at time step ( $t$ ), ( $\beta$ ) is the momentum coefficient, ( $\alpha$ ) is the learning rate, and ( $\nabla L$ ) is the gradient. Finally, the parameter is updated using the velocity:

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

Momentum helps accelerate convergence by reducing oscillations in directions where gradients change rapidly.

### 3.5 SGD with momentum Implementation using tensorflow



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import SGD

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.reshape(-1, 28 * 28)
x_test = x_test.reshape(-1, 28 * 28)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Create and compile the model
model = Sequential([
    Dense(128, activation='relu', input_shape=(28 * 28,)),
    Dense(10, activation='softmax')
])
model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9), loss='categorical_crossentropy')

# Train and evaluate the model
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.2)
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)
```



## 3.6 Explanation

**SGD with Momentum** in TensorFlow/Keras improves standard SGD by using a momentum term to accumulate previous gradients, which helps in smoothing updates and reducing fluctuations. When using `SGD(learning_rate=0.01, momentum=0.9)`, TensorFlow applies the momentum update rule internally, ensuring more stable convergence.

Momentum enhances the learning process by allowing the optimizer to carry forward useful gradient information, reducing the effect of noisy updates and leading to faster convergence in deep learning models.

## 4. Implementing the Adam Optimizer

---

### 4.1 What Is Adam?

Adam (Adaptive Moment Estimation) is a more advanced optimization algorithm that combines the benefits of momentum and adaptive learning rates. Momentum helps accelerate convergence by smoothing the updates, while adaptive learning rates adjust the step size for each parameter based on the magnitude of its gradients.

### 4.2 How Does It Work?

Adam maintains two moving averages:

1. **First Moment:** A moving average of the gradients.
2. **Second Moment:** A moving average of the squared gradients.

These moving averages are used to compute bias-corrected estimates of the gradient and its variance. The parameters are then updated using these estimates, which adaptively scale the learning rate for each parameter.

### 4.3 Adam Implementation from scratch

```
class Adam:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-9):
        self.learning_rate = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = {}
        self.v = {}
        self.t = 0
```



```

def update(self, params, grads):
    self.t += 1
    for key in params:
        if key not in self.m:
            self.m[key] = np.zeros_like(params[key])
            self.v[key] = np.zeros_like(params[key])

    # Update moving averages
    self.m[key] = self.beta1 * self.m[key] + (1 - self.beta1) * grads[key]
    self.v[key] = self.beta2 * self.v[key] + (1 - self.beta2) * (grads[key]**2)

    # Bias correction
    m_hat = self.m[key] / (1 - self.beta1 ** self.t)
    v_hat = self.v[key] / (1 - self.beta2 ** self.t)

    # Update parameters
    params[key] -= self.learning_rate * m_hat / (np.sqrt(v_hat) + self.epsilon)

```

## 4.4 Explanation

### 1. Initialization

The `Adam` class is initialized with several key hyperparameters that control its behavior. The `learning_rate` determines the step size for updating the model's parameters. A smaller learning rate leads to slower but more stable updates, while a larger learning rate can speed up training but may cause instability. The `beta1` parameter is the exponential decay rate for the first moment (mean of gradients), typically set to `0.9`. It controls how much of the past gradient information is retained. A higher value means the optimizer relies more on past gradients. The `beta2` parameter is the exponential decay rate for the second moment (variance of gradients), typically set to `0.999`. It determines how much of the past squared gradient information is retained, helping to adapt the learning rate for each parameter. The `epsilon` parameter is a small constant (e.g.,  $10^{-8}$ ) added to the denominator during parameter updates to prevent division by zero.

Additionally, the class initializes three important attributes: `self.m`, a dictionary to store the first moment (exponentially decaying average of gradients) for each parameter, which helps smooth out the gradient updates; `self.v`, a dictionary to store the second moment (exponentially decaying average of squared gradients) for each parameter, which helps to adapt the learning rate based on the magnitude of the gradients; and `self.t`, a counter to keep track of the number of updates (time steps), used for bias correction.

### 2. The update Method

The `update` method is the core of the Adam optimizer. It takes two inputs: `params`, a dictionary containing the model's parameters (e.g., weights and biases), and `grads`, a dictionary containing the gradients of the loss with respect to the parameters. If a parameter is encountered for the first time, its corresponding first moment (`self.m`) and second moment (`self.v`) are initialized to zero. This ensures that the optimizer has a starting point for tracking the moving averages of the gradients.

The first moment (`self.m`) and second moment (`self.v`) are updated using exponential moving averages. The first moment is updated as  $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ , where  $g_t$  is the gradient at the current time step. The second moment is updated as  $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ . These updates smooth out the gradients and their squares, making the optimization process more stable.

Since the moving averages are initialized to zero, they are biased toward zero during the initial time steps. To correct this bias, Adam computes bias-corrected estimates. The bias-corrected first moment is  $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$ , and the bias-corrected second moment is  $\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$ . These corrections ensure that the moving averages are accurate, especially in the early stages of training.

Finally, the parameters are updated using the bias-corrected moments:

$\theta_t = \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$ , where  $\theta_t$  is the parameter at time step  $t$ ,  $\eta$  is the learning rate, and  $\epsilon$  is a small constant to avoid division by zero. This update rule adapts the learning rate for each parameter based on the magnitude of its gradients, making Adam highly effective for training deep neural networks.

## 4.5 Adam Implementation using tensorflow

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical # Load and preprocess data

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.reshape(-1, 28 * 28)
x_test = x_test.reshape(-1, 28 * 28)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Create and compile the model
model = Sequential([Dense(128, activation='relu', input_shape=(28 * 28,)), Dense(10)])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['acc
```



```
# Train and evaluate the model
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.2)
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)
```

## 4.6 Explanation

The Adam optimizer in TensorFlow/Keras implements the Adam algorithm, which is a combination of **momentum** and **adaptive learning rates**. When you compile the model with `optimizer='adam'`, TensorFlow automatically sets up Adam to maintain two moving averages for each parameter: one for the gradients (first moment) and one for the squared gradients (second moment). These moving averages help Adam adapt the learning rate for each parameter individually, making it highly efficient.

Adam also applies **bias correction** to ensure the moving averages are accurate, especially in the early stages of training. This makes Adam robust and fast-converging, which is why it's widely used for most deep learning tasks. By dynamically adjusting the learning rate for each parameter based on the magnitude of its gradients, Adam ensures stable and efficient training, even in complex loss landscapes.

## 5. Implementing RMSProp

---

### 5.1 What Is RMSProp?

RMSProp (Root Mean Square Propagation) is another adaptive learning rate optimization algorithm. It uses an exponential moving average of squared gradients to scale the learning rate for each parameter. This helps stabilize the updates, especially in cases where the gradients vary widely.

### 5.2 How Does It Work?

RMSProp computes a moving average of the squared gradients and uses this to normalize the updates. This normalization ensures that parameters with large gradients take smaller steps, while parameters with small gradients take larger steps. This adaptive behavior helps the algorithm converge faster.

### 5.3 Python Implementation

Here's how we can implement RMSProp from scratch:



```

class RMSProp:
    def __init__(self, learning_rate=0.001, beta=0.9, epsilon=1e-8):
        self.learning_rate = learning_rate
        self.beta = beta
        self.epsilon = epsilon
        self.v = {}

    def update(self, params, grads):
        for key in params:
            if key not in self.v:
                self.v[key] = np.zeros_like(params[key])

            # Update moving average of squared gradients
            self.v[key] = self.beta * self.v[key] + (1 - self.beta) * (grads[k

            # Update parameters
            params[key] -= self.learning_rate * grads[key] / (np.sqrt(self.v[k

```

## 5.4 Explanation

### 1. Initialization

The `RMSProp` class is initialized with three key hyperparameters: `learning_rate`, `beta`, and `epsilon`. The `learning_rate` controls the step size for updating the model's parameters. A smaller learning rate means the updates are smaller and more cautious, while a larger learning rate can speed up training but might lead to instability. The `beta` parameter is a decay rate, typically set to a value like `0.9`, which controls how much of the past squared gradient information is retained. A higher value means the optimizer relies more on past gradients, while a lower value gives more weight to recent gradients. The `epsilon` parameter is a very small constant (e.g.,  $10^{-8}$ ) added to the denominator to prevent division by zero during updates. Additionally, the class initializes an empty dictionary called `self.v` to store the moving average of squared gradients for each parameter.

### 2. The update Method

The `update` method is where the actual optimization happens. It takes two inputs: `params`, which is a dictionary containing the model's parameters (like weights and biases), and `grads`, which is a dictionary containing the gradients of the loss with respect to those parameters. For each parameter, the method first checks if it has been encountered before. If not, it initializes a corresponding entry in the `self.v` dictionary to store the moving average of squared gradients, starting with zeros.

Next, the method updates the moving average of squared gradients using the formula:

$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot g_t^2$$

Here,  $v_t$  is the updated moving average,  $\beta$  is the decay rate, and  $g_t$  is the current gradient. This formula smooths out the squared gradients over time, giving more weight to recent gradients while still considering past information.

Finally, the parameters are updated using the formula:

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{g_t}{\sqrt{v_t} + \epsilon}$$

Here,  $\theta_t$  is the updated parameter,  $\eta$  is the learning rate,  $g_t$  is the current gradient,  $v_t$  is the moving average of squared gradients, and  $\epsilon$  is the small constant to avoid division by zero. This update rule adapts the learning rate for each parameter based on the magnitude of its gradients, effectively scaling down updates for parameters with large gradients and scaling up updates for parameters with small gradients. This makes RMSProp particularly effective for handling problems where the gradients vary widely in magnitude.

## 5.5 RMSProp Implementation using tensorflow

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical # Load and preprocess data

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.reshape(-1, 28 * 28)
x_test = x_test.reshape(-1, 28 * 28)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Create and compile the model
model = Sequential([Dense(128, activation='relu', input_shape=(28 * 28,)), Dense(10)])
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Train and evaluate the model
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.2)
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)
```

## 5.6 Explanation

The RMSProp optimizer in TensorFlow/Keras implements the **RMSProp algorithm**, which adapts the learning rate for each parameter based on the magnitude of its gradients. When you compile the model with `optimizer='rmsprop'` , TensorFlow sets up RMSProp to maintain a moving average of squared gradients for each parameter. This moving average is used to scale the learning rate, allowing RMSProp to handle noisy or non-stationary gradients effectively.

Unlike Adam, RMSProp does not use momentum, but it still adapts the learning rate dynamically. This makes RMSProp a good choice for problems where gradients vary significantly, as it ensures that parameters with large gradients take smaller steps, while parameters with small gradients take larger steps. This adaptive behavior helps stabilize the training process and accelerates convergence.

## 6. Comparison of Adam, RMSProp, and SGD

Feature	Adam	RMSProp	SGD
Learning Rate	Adapts for each parameter.	Adapts for each parameter.	Fixed for all parameters.
Momentum	Yes (combines momentum and RMSprop).	No (only adapts learning rate).	No (unless using SGD with momentum).
Bias Correction	Yes (corrects initialization bias).	No.	No.
Convergence Speed	Fast.	Moderate.	Slow (unless tuned well).
Robustness	Highly robust to noisy/sparse gradients.	Robust to noisy gradients.	Sensitive to learning rate and noisy gradients.
Use Case	General-purpose, works well for most problems.	Non-stationary objectives, noisy gradients.	Simple problems, small datasets.

## 7. Conclusion

---

Optimization algorithms are essential for training machine learning models. In this post, we implemented **SGD**, **Adam**, and **RMSProp** from scratch, compared them with TensorFlow/Keras implementations, and analyzed their performance. Whether you're a beginner or an expert, understanding these algorithms will help you build better models.

By experimenting with different optimizers and tuning their hyperparameters, you can significantly improve the efficiency and accuracy of your machine learning models. Each optimizer has its strengths and weaknesses, and choosing the right one depends on the specific problem you're solving.