RMSProp Optimizer in Deep Learning

Last Updated : 30 Sep, 2025

RMSProp (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm designed to improve the performance and speed of training deep learning models.

- It is a variant of the gradient descent algorithm which adapts the learning rate for each parameter individually by considering the magnitude of recent gradients for those parameters.
- This adaptive nature helps in dealing with the challenges of non-stationary objectives and sparse gradients commonly encountered in deep learning tasks.

Need of RMSProp Optimizer

RMSProp was developed to address the limitations of previous optimization methods such as SGD

Skip to content

(Stochastic Gradient Descent) and AdaGrad as SGD uses a constant learning rate which can be inefficient and AdaGrad reduces the learning rate too aggressively.

RMSProp balances by adapting the learning rates based on a moving average of squared gradients. This approach helps in maintaining a balance between efficient convergence and stability during the training process making RMSProp a widely used optimization algorithm in modern deep learning.

How RMSProp Works?

RMSProp keeps a moving average of the squared gradients to normalize the gradient updates. By doing so it prevents the learning rate from becoming too small which was a drawback in AdaGrad and ensures that the updates are appropriately scaled for each parameter. This mechanism allows RMSProp to perform well even in the presence of non-stationary objectives making it suitable for training deep learning models.

The mathematical formulation is as follows:

1. Compute the gradient g_t at time step t:

$$gt = \nabla\theta$$

2. Update the moving average of squared gradients:

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

where γ is the decay rate.

3. Update the parameter θ using the adjusted learning rate:

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

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

Parameters Used in RMSProp

- **Learning Rate (η)**: Controls the step size during the parameter updates. RMSProp typically uses a default learning rate of 0.001, but it can be adjusted based on the specific problem.
- **Decay Rate (γ)**: Determines how quickly the moving average of squared gradients decays. A

Skip to content

common default value is 0.9 which balances the contribution of recent and past gradients.

- **Epsilon (ϵ)**: A small constant added to the denominator to prevent division by zero and ensure numerical stability. A typical value for ϵ is 1e-8.

By carefully adjusting these parameters, RMSProp effectively adapts the learning rates during training, leading to faster and more reliable convergence in deep learning models.

Implementing RMSprop in Python using TensorFlow or Keras

We will use the following code line for initializing the RMSProp optimizer with hyperparameters:

> ***tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)***

- **learning_rate=0.001:** Sets the step size for weight updates. Smaller learning rates result in smaller updates, helping to fine-tune weights and prevent overshooting the minimum loss.

Skip to content

- **rho=0.9:** The discounting factor for the history of gradients, controlling the influence of past gradients on the current gradient computation.

1. Importing Libraries

We are importing libraries to implement RMSprop optimizer, handle datasets, build the model and plot results.

- tensorflow.keras for deep learning components.
- matplotlib.pyplot for visualization.

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,
Flatten
from tensorflow.keras.utils import
to_categorical
import matplotlib.pyplot as plt
```

2. Loading and Preprocessing Dataset

We load the MNIST dataset, normalize pixel values to [0,1] and one-hot encode labels.

- **mnist.load_data()** loads images and labels.

- **Normalization** improves training stability.
- **to_categorical()** converts labels to one-hot vectors.

```
(x_train, y_train), (x_test, y_test) =
mnist.load_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

3. Building the Model

We define a neural network using Sequential with input flattening and dense layers.

- Flatten converts 2D images to 1D vectors.
- Dense layers learn patterns with ReLU and softmax activations.

```
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

Skip to content

4. Compiling the Model

We compile the model using the RMSprop optimizer for adaptive learning rates, categorical cross-entropy loss for multi-class classification and track accuracy metric.

- RMSprop adjusts learning rates based on recent gradients (parameter rho controls decay rate).
- categorical_crossentropy suits one-hot encoded labels.

```
model.compile(optimizer=tf.keras.optimizers.RM
rop(learning_rate=0.001, rho=0.9),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

5. Training the Model

We train the model over 10 epochs with batch size 32 and validate on 20% of training data. validation_split monitors model performance on unseen data each epoch.

```
history = model.fit(x_train, y_train, epochs=1
                    batch_size=32,
validation_split=0.2)
```

Skip to content

Output:

```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8721 - loss: 0.4363 - val_accuracy: 0.9524 - val_loss: 0.1541
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9624 - loss: 0.1212 - val_accuracy: 0.9673 - val_loss: 0.1173
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9737 - loss: 0.0864 - val_accuracy: 0.9685 - val_loss: 0.1177
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9799 - loss: 0.0664 - val_accuracy: 0.9719 - val_loss: 0.1088
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9838 - loss: 0.0526 - val_accuracy: 0.9747 - val_loss: 0.0992
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9880 - loss: 0.0397 - val_accuracy: 0.9737 - val_loss: 0.1076
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9907 - loss: 0.0338 - val_accuracy: 0.9736 - val_loss: 0.1200
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9920 - loss: 0.0284 - val_accuracy: 0.9749 - val_loss: 0.1208
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9928 - loss: 0.0242 - val_accuracy: 0.9707 - val_loss: 0.1409
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.9933 - loss: 0.0214 - val_accuracy: 0.9758 - val_loss: 0.1398
```

Training the Model

6. Evaluating and Visualizing Results

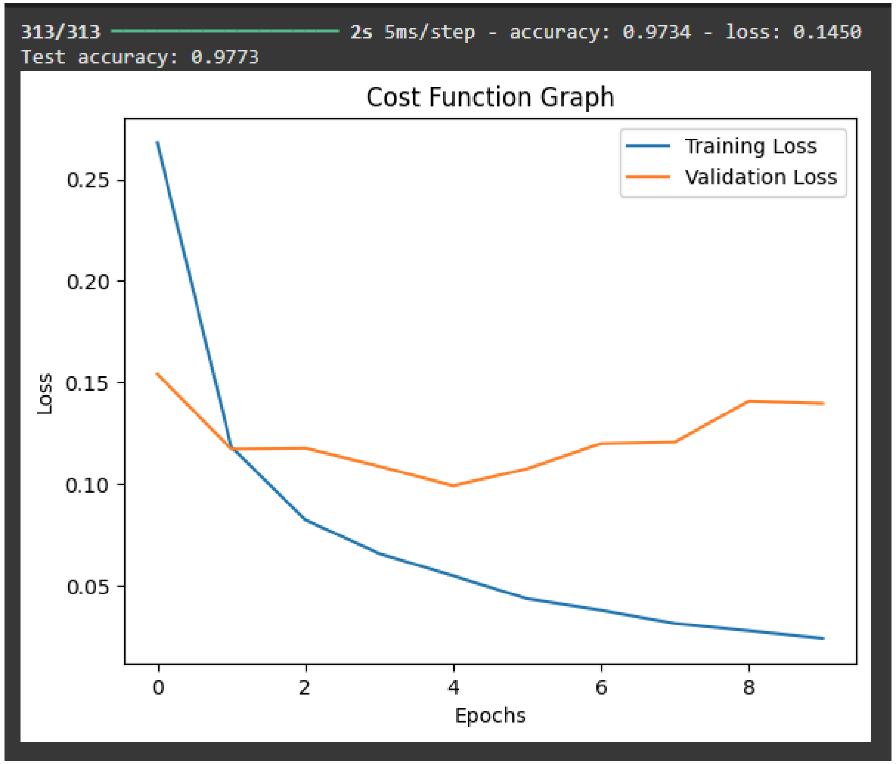
We evaluate test accuracy on unseen test data and plot training and validation loss curves to visualize learning progress.

```python
loss, accuracy = model.evaluate(x_test, y_test
print(f'Test accuracy: {accuracy:.4f}')

plt.plot(history.history['loss'],
label='Training Loss')
plt.plot(history.history['val_loss'],
label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Cost Function Graph')
plt.legend()
plt.show()
```

Output:

Skip to content



Evaluating and Visualizing Results

Advantages

- **Adaptive Learning Rates:** Adjusts learning rates for each parameter individually, optimizing updates more effectively.
- **Handles Non-Stationary Objectives:** Efficiently adapts to changing optimal parameter values over time.
- **Prevents Learning Rate Decay Problem:** Maintains optimal learning rates by using a decay rate unlike AdaGrad.

Skip to content

- **Improved Convergence Speed:** Faster convergence due to balanced and dynamic learning rates.

Disadvantages

- **Sensitivity to Hyperparameters:** Performance is sensitive to settings like decay rate and epsilon meaning it requires careful tuning.
- **Poor Performance with Sparse Data:** May struggle with sparse data, leading to slower or inconsistent convergence.

Suggested Quiz 5 Questions

Which of the following best describes the purpose of RMSProp in optimization?

A. It uses a fixed learning rate for all parameters.

B. It ignores past gradients and only uses the current gradient for updates.

C. It adapts the learning rate for each parameter based on recent gradient

Login to View Explanation 1/5 < Previous Next >

Comment

A alka1974 + Follow 3

Article Tags :

Blogathon Deep Learning AI-ML-DS

AI-ML-DS With Python

+1 More

Explore

Deep Learning Basics

Neural Networks Basics

Deep Learning Models

Deep Learning Frameworks

Model Evaluation

Skip to content

Deep Learning Projects