Importance of Optimizers in Deep Learning

Github Repository Link - https://github.com/saikumar-gaddam/Vision_Transformers-vs-CNNs-Image-Classification

1. Introduction

Optimizers play a crucial role in deep learning by determining how a neural network learns. They adjust model parameters (weights and biases) to minimize the error (loss function), helping the model make better predictions. The choice of optimizer can significantly impact:

- **Training Speed** – How fast the model converges to a good solution.
- **Model Accuracy** – How well the model generalizes to unseen data.
- **Computational Efficiency** – How much time and resources the training requires.

1.1 Why Are Optimizers Important?

Training a deep learning model involves minimizing a loss function (e.g., cross-entropy loss for classification or mean squared error for regression). The optimizer determines how weight updates are performed based on the loss gradient. A well-chosen optimizer helps:

- Avoid local minima in complex loss landscapes.
- Handle exploding or vanishing gradients in deep networks.
- Achieve faster and more stable convergence.

1.2 Gradient Descent and Its Challenges

Most optimizers are based on **Gradient Descent**, where model weights are updated using the equation:

$Wnew = Wold - \eta \cdot \nabla L$

Where:

- W = Model parameters (weights).
- η = Learning rate (step size).
- ∇L = Gradient of the loss function.

However, **vanilla gradient descent has major issues**, such as:

- **Slow convergence** for high-dimensional data.
- **Getting stuck in local minima**, making training unstable.
- **Sensitivity to learning rate** (too high leads to overshooting; too low results in slow training).

To address these challenges, advanced optimizers like **SGD, Momentum, RMSprop, and Adam** have been developed.

1.3 Types of Optimizers to Compare

This tutorial will analyze five key optimizers:

1. **Stochastic Gradient Descent (SGD)** – Simple but slow; requires careful tuning of the learning rate.
2. **Momentum-based SGD** – Uses past gradients to accelerate learning.
3. **RMSprop (Root Mean Square Propagation)** – Adapts learning rates for stable updates.
4. **Adam (Adaptive Moment Estimation)** – Combines Momentum and RMSprop for efficiency.
5. **AdamW (Weight Decay Adam)** – Improved version of Adam, widely used in modern deep learning.

2. Mathematical Understanding of Optimizers

Now that we understand the importance of optimizers, let's explore the **mathematical foundations** behind different optimization techniques. Each optimizer modifies the weight update rule to improve **learning speed, stability, and convergence**.

2.1 Stochastic Gradient Descent (SGD)

Standard Gradient Descent updates model parameters using the equation:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \nabla L$$

Where:

- W = Model parameters (weights).
- η = Learning rate (step size).
- ∇L = Gradient of the loss function.

However, **Batch Gradient Descent** computes gradients using the **entire dataset**, which is computationally expensive. **SGD (Stochastic Gradient Descent)** solves this by updating weights **after each individual data sample**, leading to:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \nabla L(W; x_i, y_i)$$

where (x_i, y_i) is a single training sample.

Advantages of SGD

- Faster updates since computations are done on **single samples** rather than the full dataset.
- Works well for **large-scale datasets**.

Disadvantages of SGD

- **Highly noisy updates**, which may cause the model to fluctuate rather than converge smoothly.
- **May get stuck in local minima** instead of reaching the optimal solution.

2.2 Momentum-Based SGD

Momentum helps SGD overcome noisy updates by **accumulating past gradients**. Instead of directly updating weights using the current gradient, it applies an **exponential moving average** of past gradients:

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

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot v_t$$

Why

where:

- v_t is the velocity term (gradient accumulation).
- β is the momentum coefficient (typically **0.9**).

Momentum Works

- Helps the optimizer **move past sharp local minima**, preventing oscillations.
- Allows **faster convergence in deep networks**.

Challenges

- Requires **careful tuning** of momentum β and learning rate η.

2.3 RMSprop (Root Mean Square Propagation)

RMSprop is an adaptive optimizer that adjusts the learning rate based on recent gradient magnitudes. It maintains a **moving average of squared gradients**:

$$S_t = \beta S_{t-1} + (1 - \beta)\nabla L^2$$

$$W_{\text{new}} = W_{\text{old}} - \frac{\eta}{\sqrt{S_t + \epsilon}} \cdot \nabla L$$

where:

- S_t is the moving average of squared gradients.
- β is the decay rate (typically **0.9**).
- ϵ is a small constant to prevent division by zero.

Why RMSprop Works

- Adapts the learning rate for each weight individually.
- Works **well for non-stationary objectives**, such as training RNNs.

Challenges

- The decay parameter β needs to be tuned for best performance.

4. Adam (Adaptive Moment Estimation)

Adam combines **Momentum and RMSprop**, using **both past gradients and squared gradients**:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla L$$

$$S_t = \beta_2 S_{t-1} + (1 - \beta_2)\nabla L^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{S}_t = \frac{S_t}{1 - \beta_2^t}$$

$$W_{\text{new}} = W_{\text{old}} - \frac{\eta}{\sqrt{\hat{S}_t + \epsilon}} \cdot \hat{m}_t$$

where:

- m_t is the **exponentially moving average** of past gradients (like Momentum).
- S_t is the **moving average of squared gradients** (like RMSprop).
- β_1 and β_2 are decay rates (commonly **0.9 and 0.999**).
- $\hat{m}_t$ and $\hat{S}_t$ are bias-corrected estimates.

Why Adam Works

- Adapts **both learning rate and momentum** dynamically.
- Works well **in most deep learning tasks** without much tuning.

Challenges

- Can sometimes **overfit due to aggressive updates**.
- Can be **sensitive to batch size**.

5. AdamW (Weight Decay Adam)

AdamW is an improved version of Adam that **fixes weight decay handling**, making it more suitable for modern architectures like **Transformers and ResNets**. The key difference is that **L2 regularization (weight decay) is decoupled from the optimization step**:

$$W_{\text{new}} = W_{\text{old}} - \frac{\eta}{\sqrt{\hat{S}_t + \epsilon}} \cdot \hat{m}_t - \lambda W_{\text{old}}$$

where λ is the **weight decay parameter**.

Why AdamW is Better

- **Fixes Adam's weight decay issues**, leading to better generalization.
- Used in **modern deep learning architectures** like BERT and Vision Transformers.

Summary of Optimizers

Optimizer	Speed	Convergence	Adaptability	Best Use Cases
SGD	Slow	Can get stuck in local minima	No	Simple models, convex optimization
Momentum	Faster than SGD	Better at escaping local minima	No	Deep learning, CNNs
RMSprop	Fast	Adapts to different gradients	Yes	RNNs, NLP tasks
Adam	Very Fast	Best for general deep learning	Yes	CNNs, RNNs, NLP, GANs
AdamW	Very Fast	Best generalization	Yes	Large-scale models, Transformers

3. Dataset Selection & Preprocessing

3.1 Dataset Selection: MNIST Handwritten Digits

For this tutorial, we have selected the **MNIST dataset**, which consists of **grayscale images of handwritten digits (0-9)**.

Why MNIST?

- Widely used in deep learning benchmarks.
- Simple but effective for testing different optimizers.
- Small dataset size, allowing fast training and comparisons.

Dataset Summary

Dataset	Training Samples	Test Samples	Image Size	Classes
MNIST	60,000	10,000	28 × 28 (Grayscale)	10 (Digits 0-9)

3.2 Data Preprocessing

Before training, we need to **prepare the dataset** by performing the following preprocessing steps:

Step 1: Loading the Dataset

We load the dataset directly from Keras:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Print dataset shapes
print(f"Training set shape: {X_train.shape}, Labels shape: {y_train.shape}")
print(f"Test set shape: {X_test.shape}, Labels shape: {y_test.shape}")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Training set shape: (60000, 28, 28), Labels shape: (60000,)
Test set shape: (10000, 28, 28), Labels shape: (10000,)
```

Step 2: Visualise the MNIST Data

```
fig, axes = plt.subplots(1, 5, figsize=(12, 3))
for i in range(5):
    axes[i].imshow(X_train[i], cmap='gray')
    axes[i].set_title(f"Label: {y_train[i]}")
    axes[i].axis("off")

plt.show()
```

Step 3: Data Preprocesssing

Normalization

Neural networks work best when input values are in the range **0 to 1** instead of **0 to 255**. We normalize the pixel values by dividing by **255.0**

Reshaping the Data

Since CNNs require **4D input tensors** (batch, height, width, channels), we reshape the dataset:

One-Hot Encoding of Labels

The labels are integers (0-9), but we need **one-hot encoded vectors** for multi-class classification

```python
# Import necessary libraries
from tensorflow.keras.utils import to_categorical

# Step 1: Normalize pixel values to range [0, 1]
X_train = X_train.astype("float32") / 255.0
X_test = X_test.astype("float32") / 255.0

# Step 2: Reshape images to (28, 28, 1) to match CNN input requirements
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

# Step 3: Convert labels to one-hot encoding
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# Step 4: Print the new shapes after preprocessing
print("Dataset after Preprocessing:")
print(f"- Training set shape: {X_train.shape}, Labels shape: {y_train.shape}")
print(f"- Test set shape: {X_test.shape}, Labels shape: {y_test.shape}")
```

```
Dataset after Preprocessing:
- Training set shape: (60000, 28, 28, 1), Labels shape: (60000, 10)
- Test set shape: (10000, 28, 28, 1), Labels shape: (10000, 10)
```

Final Preprocessing Summary

After preprocessing, the dataset is structured as follows:

- **Training Images**: (60,000, 28, 28, 1)
- **Test Images**: (10,000, 28, 28, 1)
- **Training Labels**: (60,000, 10) (one-hot encoded)
- **Test Labels**: (10,000, 10) (one-hot encoded)

4. Model Implementation

4.1 Building the CNN Model

To effectively classify the MNIST handwritten digits, we implement a **Convolutional Neural Network (CNN)**. This model consists of:

- **Convolutional Layers**: Extract spatial features from images.
- **MaxPooling Layers**: Reduce spatial dimensions while retaining key features.
- **Flatten Layer**: Converts the extracted features into a vector for classification.
- **Dense Layers**: Fully connected layers for learning high-level patterns.
- **Output Layer**: A softmax layer with 10 neurons (one for each digit 0-9).

Step 1: Define the CNN Model

The model architecture includes:

- **Conv2D Layer 1**: 32 filters, **(3×3) kernel**, ReLU activation.
- **MaxPooling Layer 1**: Pool size **(2×2)**.
- **Conv2D Layer 2**: 64 filters, **(3×3) kernel**, ReLU activation.

- **MaxPooling Layer 2**: Pool size **(2×2)**.
- **Flatten Layer**: Converts 2D feature maps into a 1D vector.
- **Fully Connected Dense Layer**: 128 neurons, ReLU activation.
- **Output Layer**: 10 neurons with softmax activation for classification.

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Step 1: Define the CNN model
def create_cnn_model():
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
        MaxPooling2D((2,2)),
        Conv2D(64, (3,3), activation='relu'),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')  # Output layer for 10 classes
    ])
    return model

# Step 2: Print the model summary
model = create_cnn_model()
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: D
(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
dense_1 (Dense)	(None, 10)	1,290

```
Total params: 225,034 (879.04 KB)
Trainable params: 225,034 (879.04 KB)
Non-trainable params: 0 (0.00 B)
```

4.2 Model Compilation & Training

Before training, we compile the model with different optimizers to compare their effectiveness:

- **Loss Function**: categorical_crossentropy (used for multi-class classification).
- **Metrics**: accuracy (to measure classification performance).
- **Optimizers**: We will train using **SGD, Momentum, RMSprop, Adam, and AdamW**.

```python
from tensorflow.keras.optimizers import SGD, RMSprop, Adam, AdamW
import time

# Define a function to train the model with a specific optimizer
def train_model_with_optimizer(optimizer, optimizer_name, epochs=15):
    model = create_cnn_model()  # Recreate the model for each optimizer
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

    print(f"\nTraining Model with {optimizer_name} Optimizer...\n")
    start_time = time.time()  # Track training time
    history = model.fit(X_train, y_train, epochs=epochs, batch_size=64, validation_data=(X_test, y_test), verbose=1)
    end_time = time.time()

    training_time = end_time - start_time
    print(f"\n{optimizer_name} Training Time: {training_time:.2f} seconds\n")

    return history
```

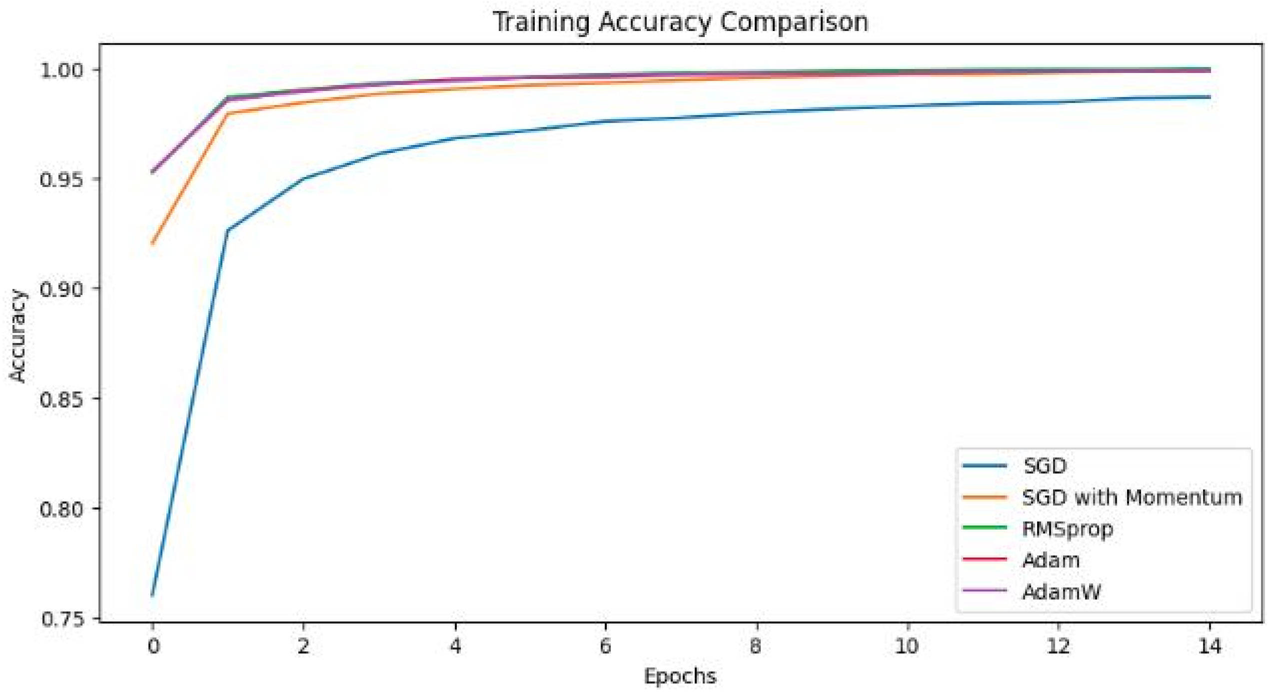
```python
# Train models with different optimizers
history_sgd = train_model_with_optimizer(SGD(learning_rate=0.01), "SGD")
history_momentum = train_model_with_optimizer(SGD(learning_rate=0.01, momentum=0.9), "SGD with Momentum")
history_rmsprop = train_model_with_optimizer(RMSprop(learning_rate=0.001), "RMSprop")
history_adam = train_model_with_optimizer(Adam(learning_rate=0.001), "Adam")
history_adamw = train_model_with_optimizer(AdamW(learning_rate=0.001), "AdamW")
```

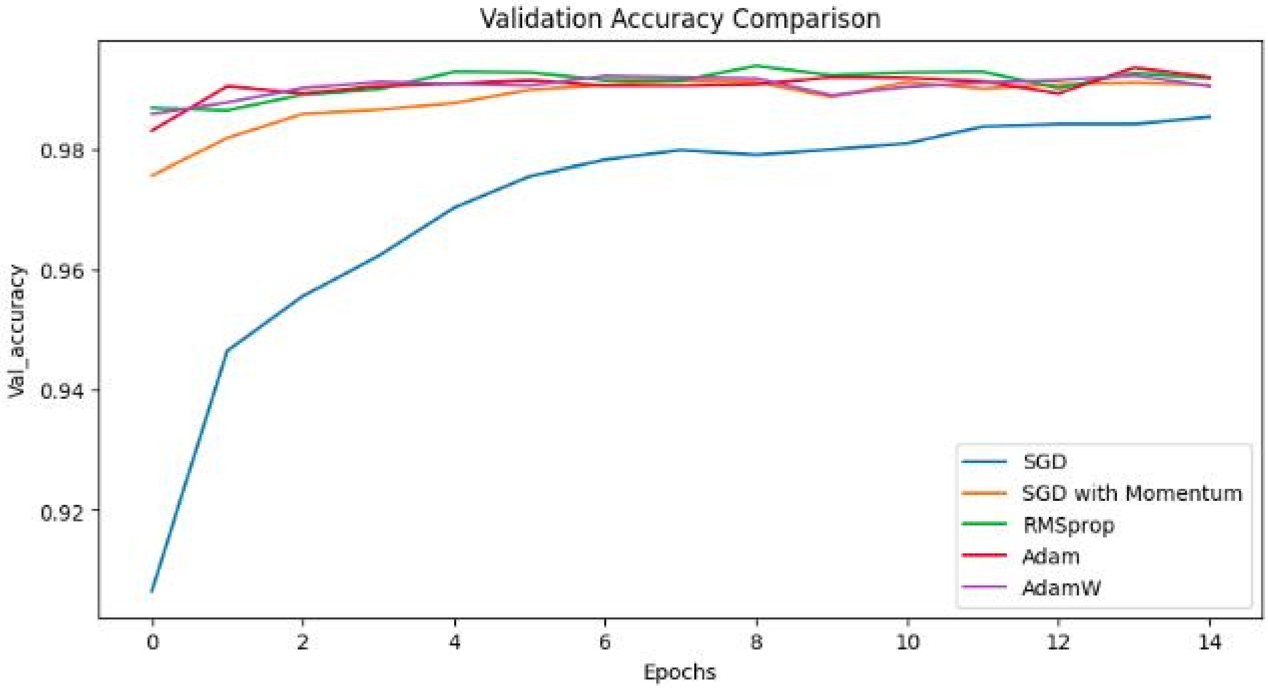
```
Training Model with SGD Optimizer...

Epoch 1/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.5635 - loss: 1.4530 - val_accuracy: 0.9066 - val_loss: 0.3053
Epoch 2/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9174 - loss: 0.2731 - val_accuracy: 0.9465 - val_loss: 0.1763
Epoch 3/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9464 - loss: 0.1789 - val_accuracy: 0.9556 - val_loss: 0.1449
Epoch 4/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9593 - loss: 0.1383 - val_accuracy: 0.9623 - val_loss: 0.1179
Epoch 5/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9681 - loss: 0.1076 - val_accuracy: 0.9703 - val_loss: 0.0960
Epoch 6/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9720 - loss: 0.0915 - val_accuracy: 0.9755 - val_loss: 0.0824
Epoch 7/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9753 - loss: 0.0863 - val_accuracy: 0.9782 - val_loss: 0.0702
Epoch 8/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 41s 25ms/step - accuracy: 0.9767 - loss: 0.0751 - val_accuracy: 0.9798 - val_loss: 0.0659
Epoch 9/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9797 - loss: 0.0681 - val_accuracy: 0.9790 - val_loss: 0.0647
Epoch 10/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9815 - loss: 0.0628 - val_accuracy: 0.9799 - val_loss: 0.0686
Epoch 11/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9829 - loss: 0.0581 - val_accuracy: 0.9809 - val_loss: 0.0623
Epoch 12/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9839 - loss: 0.0530 - val_accuracy: 0.9837 - val_loss: 0.0520
Epoch 13/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 22s 24ms/step - accuracy: 0.9838 - loss: 0.0511 - val_accuracy: 0.9841 - val_loss: 0.0504
Epoch 14/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9856 - loss: 0.0475 - val_accuracy: 0.9841 - val_loss: 0.0482
Epoch 15/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9870 - loss: 0.0438 - val_accuracy: 0.9853 - val_loss: 0.0471

SGD Training Time: 366.34 seconds


Training Model with SGD with Momentum Optimizer...

Epoch 1/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.8102 - loss: 0.6237 - val_accuracy: 0.9756 - val_loss: 0.0762
Epoch 2/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9770 - loss: 0.0730 - val_accuracy: 0.9818 - val_loss: 0.0529
Epoch 3/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9835 - loss: 0.0509 - val_accuracy: 0.9858 - val_loss: 0.0428
Epoch 4/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9885 - loss: 0.0377 - val_accuracy: 0.9865 - val_loss: 0.0401
Epoch 5/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9903 - loss: 0.0304 - val_accuracy: 0.9876 - val_loss: 0.0369
Epoch 6/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 41s 25ms/step - accuracy: 0.9920 - loss: 0.0260 - val_accuracy: 0.9898 - val_loss: 0.0315
Epoch 7/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9937 - loss: 0.0192 - val_accuracy: 0.9906 - val_loss: 0.0287
Epoch 8/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9948 - loss: 0.0153 - val_accuracy: 0.9914 - val_loss: 0.0293
Epoch 9/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9960 - loss: 0.0133 - val_accuracy: 0.9911 - val_loss: 0.0276
Epoch 10/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9966 - loss: 0.0111 - val_accuracy: 0.9886 - val_loss: 0.0340
Epoch 11/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9972 - loss: 0.0097 - val_accuracy: 0.9911 - val_loss: 0.0339
Epoch 12/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9976 - loss: 0.0085 - val_accuracy: 0.9900 - val_loss: 0.0347
Epoch 13/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9986 - loss: 0.0054 - val_accuracy: 0.9907 - val_loss: 0.0318
Epoch 14/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9982 - loss: 0.0056 - val_accuracy: 0.9910 - val_loss: 0.0308
Epoch 15/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9990 - loss: 0.0036 - val_accuracy: 0.9906 - val_loss: 0.0344

SGD with Momentum Training Time: 365.15 seconds
```

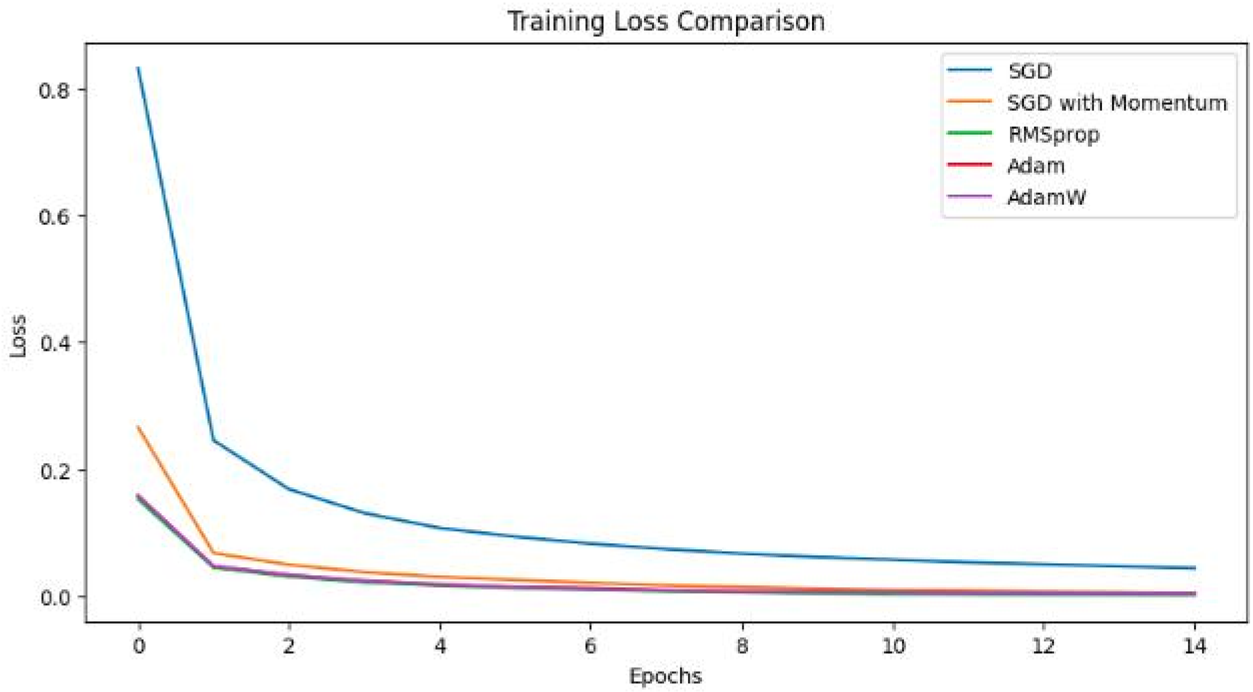
```
Training Model with RMSprop Optimizer...

Epoch 1/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.8889 - loss: 0.3525 - val_accuracy: 0.9868 - val_loss: 0.0431
Epoch 2/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9856 - loss: 0.0488 - val_accuracy: 0.9864 - val_loss: 0.0377
Epoch 3/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9896 - loss: 0.0306 - val_accuracy: 0.9890 - val_loss: 0.0329
Epoch 4/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9933 - loss: 0.0215 - val_accuracy: 0.9899 - val_loss: 0.0317
Epoch 5/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9948 - loss: 0.0161 - val_accuracy: 0.9928 - val_loss: 0.0221
Epoch 6/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9967 - loss: 0.0106 - val_accuracy: 0.9927 - val_loss: 0.0255
Epoch 7/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9972 - loss: 0.0082 - val_accuracy: 0.9914 - val_loss: 0.0301
Epoch 8/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 40s 24ms/step - accuracy: 0.9975 - loss: 0.0078 - val_accuracy: 0.9913 - val_loss: 0.0369
Epoch 9/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9980 - loss: 0.0061 - val_accuracy: 0.9938 - val_loss: 0.0284
Epoch 10/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9988 - loss: 0.0050 - val_accuracy: 0.9922 - val_loss: 0.0363
Epoch 11/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9992 - loss: 0.0024 - val_accuracy: 0.9927 - val_loss: 0.0346
Epoch 12/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9993 - loss: 0.0031 - val_accuracy: 0.9928 - val_loss: 0.0382
Epoch 13/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9994 - loss: 0.0016 - val_accuracy: 0.9901 - val_loss: 0.0602
Epoch 14/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9995 - loss: 0.0015 - val_accuracy: 0.9926 - val_loss: 0.0397
Epoch 15/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9995 - loss: 0.0014 - val_accuracy: 0.9917 - val_loss: 0.0519

RMSprop Training Time: 365.02 seconds


Training Model with Adam Optimizer...

Epoch 1/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 25s 26ms/step - accuracy: 0.8903 - loss: 0.3762 - val_accuracy: 0.9838 - val_loss: 0.0517
Epoch 2/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9841 - loss: 0.0502 - val_accuracy: 0.9904 - val_loss: 0.0300
Epoch 3/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9897 - loss: 0.0328 - val_accuracy: 0.9892 - val_loss: 0.0317
Epoch 4/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9932 - loss: 0.0224 - val_accuracy: 0.9904 - val_loss: 0.0298
Epoch 5/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9947 - loss: 0.0175 - val_accuracy: 0.9908 - val_loss: 0.0285
Epoch 6/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9966 - loss: 0.0116 - val_accuracy: 0.9914 - val_loss: 0.0287
Epoch 7/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9963 - loss: 0.0108 - val_accuracy: 0.9905 - val_loss: 0.0335
Epoch 8/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 42s 26ms/step - accuracy: 0.9976 - loss: 0.0081 - val_accuracy: 0.9905 - val_loss: 0.0329
Epoch 9/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 40s 25ms/step - accuracy: 0.9976 - loss: 0.0065 - val_accuracy: 0.9907 - val_loss: 0.0361
Epoch 10/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9978 - loss: 0.0060 - val_accuracy: 0.9919 - val_loss: 0.0278
Epoch 11/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9983 - loss: 0.0050 - val_accuracy: 0.9918 - val_loss: 0.0344
Epoch 12/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9983 - loss: 0.0048 - val_accuracy: 0.9912 - val_loss: 0.0367
Epoch 13/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9989 - loss: 0.0038 - val_accuracy: 0.9892 - val_loss: 0.0454
Epoch 14/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9981 - loss: 0.0053 - val_accuracy: 0.9935 - val_loss: 0.0338
Epoch 15/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9988 - loss: 0.0036 - val_accuracy: 0.9919 - val_loss: 0.0396

Adam Training Time: 389.34 seconds
```

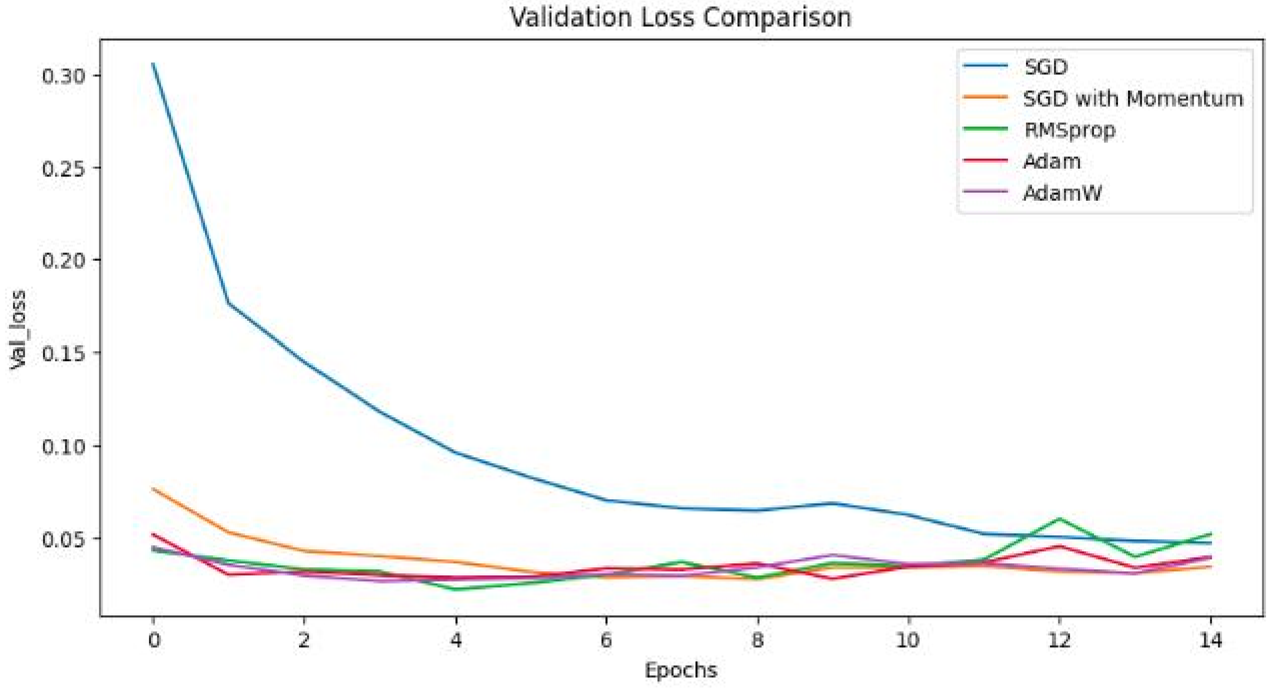
```
Training Model with AdamW Optimizer...

Epoch 1/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 26s 26ms/step - accuracy: 0.8885 - loss: 0.3719 - val_accuracy: 0.9858 - val_loss: 0.0449
Epoch 2/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 26ms/step - accuracy: 0.9852 - loss: 0.0479 - val_accuracy: 0.9877 - val_loss: 0.0354
Epoch 3/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9888 - loss: 0.0338 - val_accuracy: 0.9901 - val_loss: 0.0296
Epoch 4/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9924 - loss: 0.0227 - val_accuracy: 0.9911 - val_loss: 0.0266
Epoch 5/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 40s 25ms/step - accuracy: 0.9947 - loss: 0.0170 - val_accuracy: 0.9908 - val_loss: 0.0273
Epoch 6/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9963 - loss: 0.0121 - val_accuracy: 0.9905 - val_loss: 0.0281
Epoch 7/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9970 - loss: 0.0101 - val_accuracy: 0.9921 - val_loss: 0.0304
Epoch 8/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 24s 25ms/step - accuracy: 0.9977 - loss: 0.0073 - val_accuracy: 0.9919 - val_loss: 0.0294
Epoch 9/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9983 - loss: 0.0058 - val_accuracy: 0.9917 - val_loss: 0.0348
Epoch 10/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9980 - loss: 0.0058 - val_accuracy: 0.9889 - val_loss: 0.0407
Epoch 11/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9979 - loss: 0.0063 - val_accuracy: 0.9903 - val_loss: 0.0357
Epoch 12/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 24ms/step - accuracy: 0.9986 - loss: 0.0044 - val_accuracy: 0.9910 - val_loss: 0.0363
Epoch 13/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9986 - loss: 0.0039 - val_accuracy: 0.9914 - val_loss: 0.0332
Epoch 14/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9988 - loss: 0.0036 - val_accuracy: 0.9922 - val_loss: 0.0307
Epoch 15/15
938/938 ━━━━━━━━━━━━━━━━━━━━ 23s 25ms/step - accuracy: 0.9993 - loss: 0.0018 - val_accuracy: 0.9904 - val_loss: 0.0391

AdamW Training Time: 370.29 seconds
```

```
import matplotlib.pyplot as plt

# Function to plot accuracy and loss curves
def plot_results(histories, title, metric):
    plt.figure(figsize=(10, 5))

    for optimizer_name, history in histories.items():
        plt.plot(history.history[metric], label=f"{optimizer_name}")

    plt.title(title)
    plt.xlabel("Epochs")
    plt.ylabel(metric.capitalize())
    plt.legend()
    plt.show()

# Store all training histories in a dictionary
histories = {
    "SGD": history_sgd,
    "SGD with Momentum": history_momentum,
    "RMSprop": history_rmsprop,
    "Adam": history_adam,
    "AdamW": history_adamw
}

# Plot Accuracy Curves
plot_results(histories, "Training Accuracy Comparison", "accuracy")
plot_results(histories, "Validation Accuracy Comparison", "val_accuracy")

# Plot Loss Curves
plot_results(histories, "Training Loss Comparison", "loss")
plot_results(histories, "Validation Loss Comparison", "val_loss")
```




Validation Accuracy Comparison
Val_accuracy
0.98
0.96
0.94
0.92
0
2
4
6
8
10
12
14
Epochs
SGD
SGD with Momentum
RMSprop
Adam
AdamW


Training Loss Comparison
Loss
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12
14
Epochs
SGD
SGD with Momentum
RMSprop
Adam
AdamW



5. Visualizing and Comparing Optimizer Performance

To **analyze the effectiveness of each optimizer**, we will:

1. Plot Training Accuracy vs. Epochs
2. Plot Validation Accuracy vs. Epochs
3. Plot Training Loss vs. Epochs
4. Plot Validation Loss vs. Epochs
5. Compare Total Training Time for Each Optimizer

5.1 Performance Analysis & Results Interpretation

Now that we have generated the accuracy and loss curves for different optimizers, let's analyze their performance.

1. Training Accuracy Comparison

Observations:

- **Adam, AdamW, RMSprop, and SGD with Momentum achieved high accuracy (~99%) within a few epochs.**
- **SGD took longer to converge** but eventually reached **~98% accuracy**.
- **Adam and AdamW reached peak accuracy the fastest**, making them the most efficient.

Conclusion:
Adam and AdamW optimizers provide the fastest and most stable accuracy improvements.

2. Validation Accuracy Comparison

Observations:

- **Adam, AdamW, and RMSprop consistently maintained higher validation accuracy.**
- **SGD showed a steady improvement but lagged behind other optimizers.**

- **SGD with Momentum performed better than standard SGD but still took more epochs to stabilize.**

Conclusion:
Adam and AdamW offer the best **generalization** on validation data, while RMSprop also performs well.

3. Training Loss Comparison

Observations:

- **Adam, AdamW, and RMSprop had the fastest loss reduction, reaching near-zero loss quickly.**
- **SGD showed the slowest reduction in loss, requiring more epochs to improve.**
- **SGD with Momentum helped accelerate loss reduction compared to plain SGD.**

Conclusion:
Adaptive optimizers (Adam, AdamW, and RMSprop) converge **faster** and more **smoothly** than traditional gradient descent.

4. Validation Loss Comparison

Observations:

- **SGD had the highest validation loss initially but gradually improved.**
- **Adam, AdamW, and RMSprop maintained the lowest validation loss.**
- **SGD with Momentum improved over standard SGD but still fluctuated slightly.**

Conclusion:
AdamW and Adam consistently resulted in **lower validation loss**, suggesting better stability and generalization.

5.2 Final Optimizer Comparison

Optimizer	Speed	Convergence	Stability	Best Use Cases
SGD	Slow	Poor	Stable but requires tuning	Basic models, convex problems
SGD with Momentum	Faster than SGD	Medium	More stable than SGD	CNNs, image recognition
RMSprop	Fast	Good	Slight fluctuations	RNNs, NLP tasks
Adam	Very Fast	Best	Highly stable	Most deep learning applications
AdamW	Very Fast	Best	Most stable	Large-scale architectures (e.g., Transformers, ResNets)

6. Conclusion

6.1 Best Practices & Recommendations: When to Use Which Optimizer?

The choice of optimizer depends on the model, dataset, and training goals. Below is a quick guide:

Optimizer	Best For	Pros	Cons
SGD	Simple models, convex problems	Computationally efficient	Slow convergence
SGD + Momentum	CNNs, deep networks	Faster convergence, reduces oscillations	Requires tuning
RMSprop	RNNs, NLP tasks	Adaptive learning rates	May be unstable
Adam	Most deep learning tasks	Fast, works well in general	Can overfit
AdamW	Large-scale models (Transformers, ResNets)	Best generalization	Slightly slower than Adam

6.2 Optimizer Selection Guidelines

- **Use Adam or AdamW** for most deep learning applications.
- **Use RMSprop** for RNNs and NLP tasks.
- **Use SGD with Momentum** if adaptive optimizers are not needed.
- **Use AdamW** for large-scale architectures requiring strong generalization.

6.3 Key Takeaways

- **Adam and AdamW** are the most efficient optimizers, achieving high accuracy quickly.
- **SGD is slow but stable**, while **SGD with Momentum speeds up training.**
- **RMSprop works well for recurrent networks.**
- **AdamW generalizes better than Adam**, making it ideal for modern architectures.

Selecting the right optimizer depends on your model's needs. **Adam and AdamW are the best choices for most deep learning tasks.**