Optimizers

Batch gradient descent

Gradient update rule: BGD uses the data of the entire training set to calculate the gradient of the cost function to the parameters:

Disadvantages:

Because this method calculates **the gradient for the entire data set in one update, the calculation is very slow**, it will be very tricky to encounter a large number of data sets, and you cannot invest in new data to update the model in real time.

We will define an iteration number epoch in advance, first calculate the gradient vector params_grad, and then update the parameter params along the direction of the gradient. The learning rate determines how big we take each step.

Batch gradient descent can converge to a global minimum for convex functions and to a local minimum for non-convex functions.

SGD (Stochastic gradient descent)

Gradient update rule: Compared with BGD's calculation of gradients with all data at one time, SGD updates the gradient of each sample with each update.

```
x += - learning_rate * dx
```

where x is a parameter, dx is the gradient and learning rate is constant

For large data sets, there may be similar samples, so BGD calculates the gradient. **There will be redundancy, and SGD is updated only once, there is no redundancy, it is faster, and new samples can be added.**

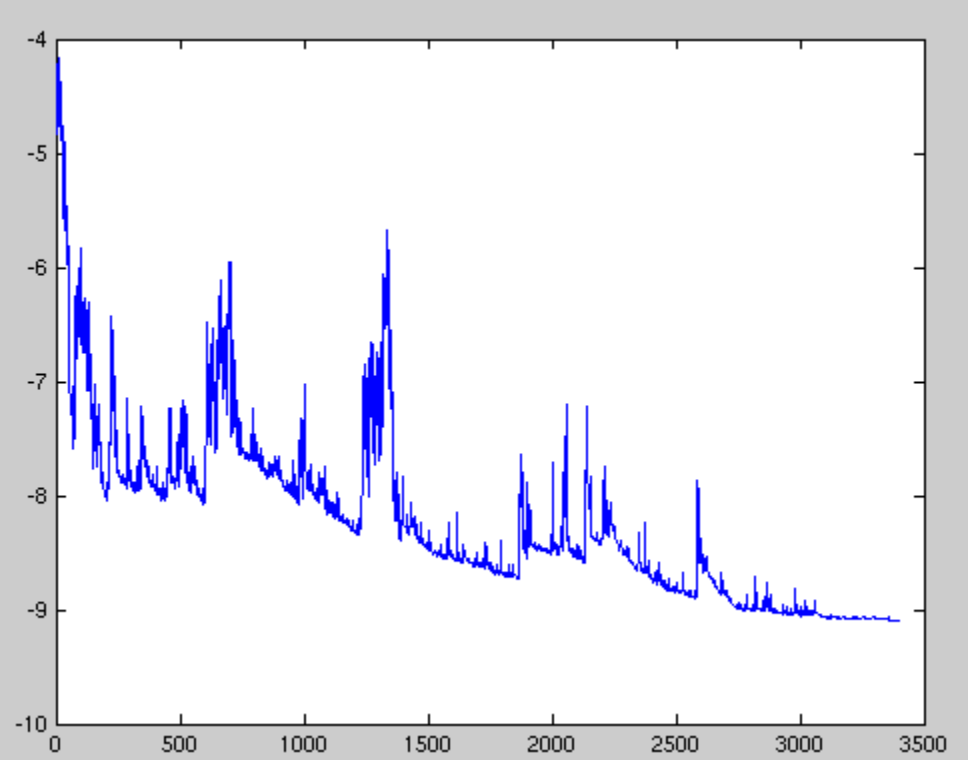


Figure :- Fluctuations in SGD

Disadvantages: However, because SGD is updated more frequently, the cost function will have severe oscillations. BGD can converge to a local minimum, of course, the oscillation of SGD may jump to a better local minimum.

When we decrease the learning rate slightly, the convergence of SGD and BGD is the same.

Mini-batch gradient descent

Gradient update rule:

MBGD uses a small batch of samples, that is, n samples to calculate each time. In this way, it can reduce the variance when the parameters are updated, and the convergence is more stable. It can make full use of the highly optimized matrix operations in the deep learning library for more efficient gradient calculations.

The difference from SGD is that each cycle does not act on each sample, but a batch with n samples.

> Setting value of hyper-parameters: n Generally value is 50 ~ 256

Cons:

- Mini-batch gradient descent does not guarantee good convergence,
- If the learning rate is too small, the convergence rate will be slow. If it is too large, the loss function will oscillate or even deviate at the minimum value. One measure is to set a **larger learning rate**. When the change between two iterations is lower than a certain threshold, the learning rate is reduced.

However, the setting of this threshold needs to be written in advance adapt to the characteristics of the data set.

In addition, this method is to apply the **same learning rate** to all parameter updates. If our data is sparse, we would prefer to update the features with lower frequency.

In addition, for **non-convex functions**, it is also necessary to avoid trapping at the local minimum or saddle point, because the error around the saddle point is the same, the gradients of all dimensions are close to 0, and SGD is easily trapped here.

Saddle points are the curves, surfaces, or hyper surfaces of a saddle point neighborhood of a smooth function are located on different sides of a tangent to this point. For example, this two-dimensional figure looks like a saddle: it curves up in the x-axis direction and down in the y-axis direction, and the saddle point is (0,0).

Momentum

One disadvantage of the SGD method is that its update direction depends entirely on the current batch, so its update is very unstable. A simple way to solve this problem is to introduce momentum.

Momentum is momentum, which simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is

Adagrad

Adagrad is an algorithm for gradient-based optimization which adapts the learning rate to the parameters, using low learning rates for parameters associated with frequently occurring features, and using high learning rates for parameters associated with infrequent features.

So, it is well-suited for dealing with sparse data.

But the same update rate may not be suitable for all parameters. For example, some parameters may have reached the stage where only fine-tuning is needed, but some parameters need to be adjusted a lot due to the small number of corresponding samples.

Adagrad proposed this problem, an algorithm that adaptively assigns different learning rates to various parameters among them. The implication is that for each parameter, as its total distance updated increases, its learning rate also slows.

> **GloVe word embedding uses adagrad where infrequent words required a greater update and frequent words require smaller updates.**

> **Adagrad eliminates the need to manually tune the learning rate.**

Adadelta

There are three problems with the Adagrad algorithm

- The learning rate is monotonically decreasing.
- The learning rate in the late training period is very small.
- It requires manually setting a global initial learning rate.

> **Adadelta is an extension of Adagrad and it also tries to reduce Adagrad's aggressive, monotonically reducing the learning rate.**

> It does this by restricting the window of the past accumulated gradient to some fixed size of w. Running average at time t then depends on the previous average and the current gradient.

> In Adadelta we do not need to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient.

RMSProp

The full name of RMSProp algorithm is called **Root Mean Square Prop**, which is an adaptive learning rate optimization algorithm proposed by Geoff Hinton.

> RMSProp tries to resolve Adagrad's radically diminishing learning rates by using a moving average of the squared gradient. It utilizes the magnitude of the recent gradient descents to normalize the gradient.

Adagrad will accumulate all previous gradient squares, and RMSprop just calculates the corresponding average value, so it can alleviate the problem that the learning rate of the Adagrad algorithm drops quickly.

The difference is that RMSProp calculates the **differential squared weighted average of the gradient** . This method is beneficial to eliminate the direction of large swing amplitude, and is used to correct the swing amplitude, so that the swing amplitude in each dimension is smaller. On the other hand, it also makes the network function converge faster.

> In RMSProp learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.

> RMSProp divides the learning rate by the average of the exponential decay of squared gradients

Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop.

> Adam also keeps an exponentially decaying average of past gradients, similar to momentum.

Adam can be viewed as a combination of Adagrad and RMSprop,(Adagrad) which works well on sparse gradients and (RMSProp) which works well in online and non-stationary settings respectively.

Adam implements the **exponential moving average of the gradients** to scale the learning rate instead of a simple average as in Adagrad. It keeps an exponentially decaying average of past gradients.

Adam is computationally efficient and has very less memory requirement.

Adam optimizer is one of the most popular and famous gradient descent optimization

Comparisons

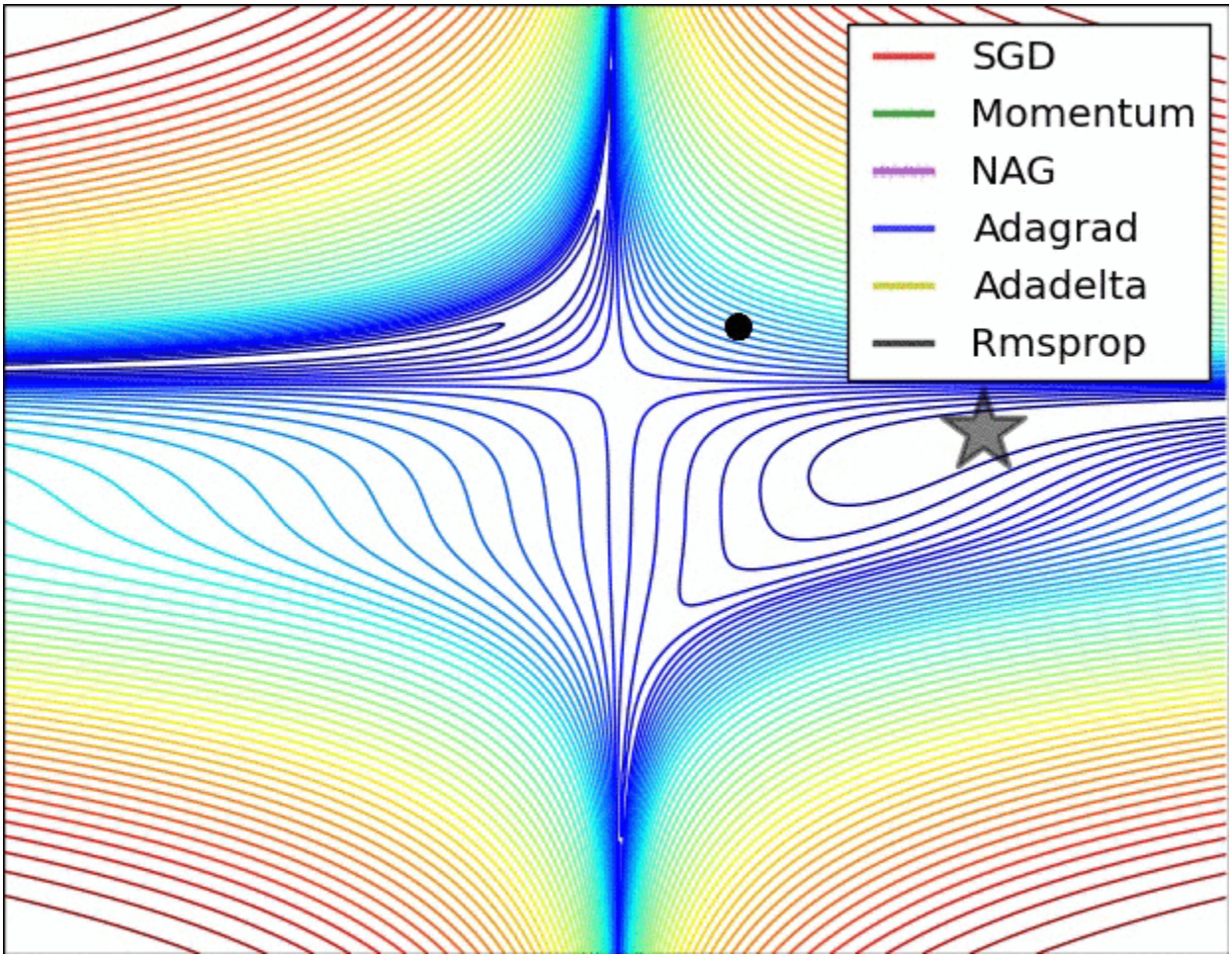


Figure :- SGD optimization on loss surface contours

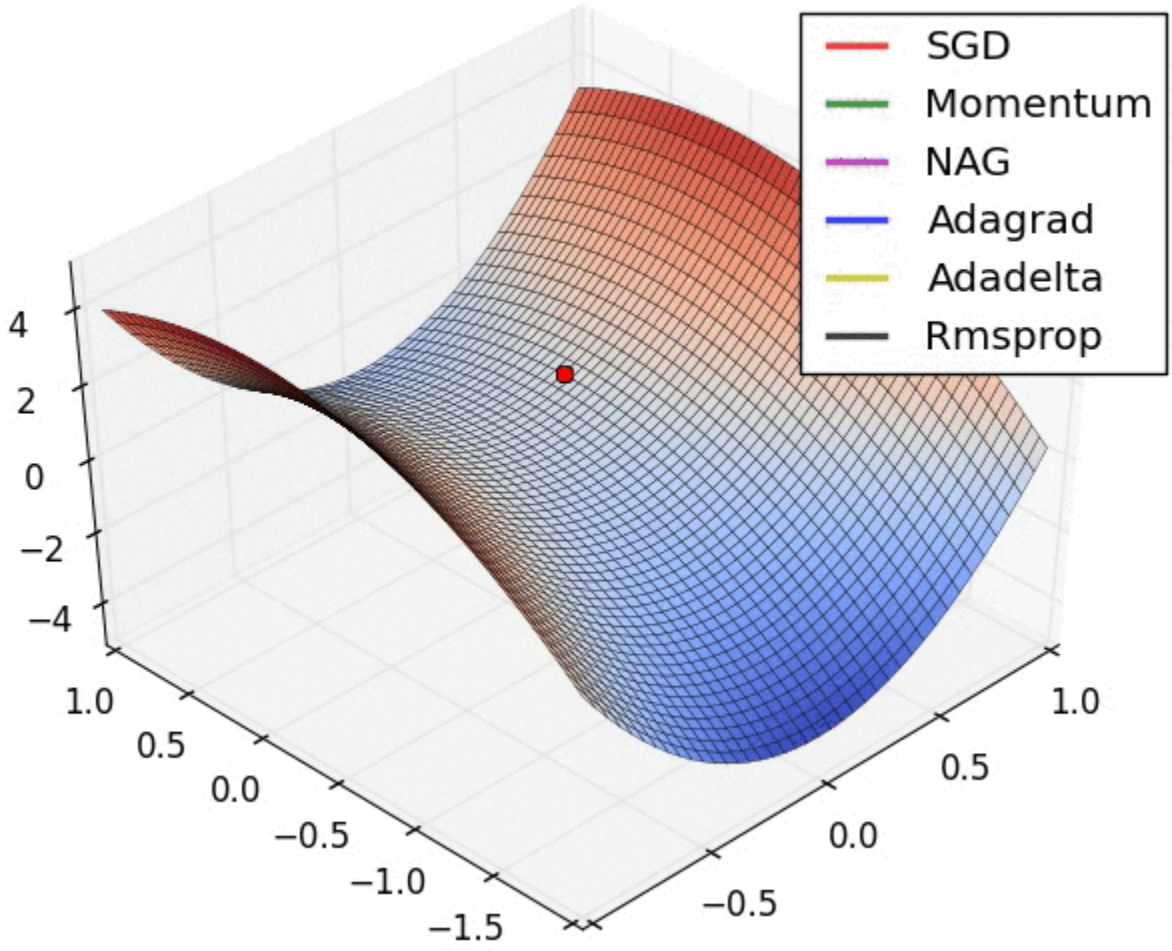


Figure :- SGD optimization on saddle point

How to choose optimizers?

- If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam.
- RMSprop, Adadelta, Adam have similar effects in many cases.
- Adam just added bias-correction and momentum on the basis of RMSprop,
- As the gradient becomes sparse, Adam will perform better than RMSprop.

In [ ]:

Assignment Question

Q1: What is the role of optimization algorithms in artificial neural networks? Why are they necessary?

A A1: Optimization algorithms play a crucial role in artificial neural networks as they are responsible for updating the network's parameters during the training process. The main objective of training a neural network is to minimize the loss function, which measures the difference between the predicted outputs and the actual targets. Optimization algorithms are necessary because they determine how the network should adjust its weights and biases to reach the optimal configuration that minimizes the loss and improves the model's performance.

Q2: Explain the concept of gradient descent and its variants. Discuss their differences and tradeoffs in terms of convergence speed and memory requirements.

A A2: Gradient descent is a popular optimization algorithm used in training neural networks. It works by calculating the gradient of the loss function with respect to the model parameters and then updating the parameters in the opposite direction of the gradient to minimize the loss. The basic variants of gradient descent include:

- Batch Gradient Descent: It computes the gradient using the entire training dataset, making it computationally expensive and memory-intensive. However, it provides a more stable convergence path, leading to better convergence.
- Stochastic Gradient Descent (SGD): This variant randomly selects one training sample at a time to compute the gradient, which reduces memory requirements but introduces more noise and oscillations during training. It converges faster in many cases but might not reach the global minimum.
- Mini-batch Gradient Descent: It strikes a balance between batch and stochastic gradient descent. It randomly samples a small subset (mini-batch) of the training data to compute the gradient, combining the advantages of both batch and SGD. It is widely used due to its efficiency and convergence properties.

Q3: Describe the challenges associated with traditional gradient descent optimization methods (e.g., slow convergence, local minima). How do modern optimizers address these challenges?

A A3: Traditional gradient descent methods, such as Batch Gradient Descent and Stochastic Gradient Descent, face several challenges during training:

- Slow Convergence: Traditional methods may take a long time to converge to the optimal solution, especially when dealing with complex and high-dimensional data.
- Local Minima: They can get trapped in local minima, preventing the model from finding the global minimum of the loss function.

Modern optimization algorithms address these challenges in the following ways:

- Momentum: Momentum is a concept in optimization that accelerates the convergence by adding a fraction of the previous update direction to the current update direction. This helps in overcoming slow convergence and escaping shallow local minima.
- Adaptive Learning Rates: Modern optimizers use adaptive learning rate techniques that adjust the learning rate during training to control the step size in parameter updates. This approach enables faster convergence by taking larger steps when the gradients are steep and smaller steps in flatter regions.

Q4: Discuss the concepts of momentum and learning rate in the context of optimization algorithms. How do they impact convergence and model performance?

A A4: Momentum and learning rate are essential concepts in optimization algorithms:

- Momentum: Momentum introduces inertia to the parameter updates, helping the optimizer to continue moving in the same direction as previous updates. This accelerates convergence and smoothes the optimization path, reducing oscillations. It can help escape local minima and speed up convergence, especially in areas with high curvature.
- Learning Rate: The learning rate controls the step size taken during parameter updates. A large learning rate may lead to overshooting and unstable updates, while a small learning rate may slow down convergence. It needs to be carefully tuned to find the right balance between fast convergence and avoiding divergence.
- Both momentum and learning rate significantly impact convergence and model performance. Properly tuned momentum can help the optimizer navigate complex loss surfaces, while a suitable learning rate is crucial for achieving fast and stable convergence without overshooting the optimal solution.

Part 2: Optimizer Techniques

Q1: Explain the concept of Stochastic Gradient Descent (SGD) and its advantages compared to traditional gradient descent. Discuss its limitations and scenarios where it is most suitable.

A A1: Stochastic Gradient Descent (SGD) is an optimization technique that computes the gradient and updates the model's parameters for each individual training sample. Its advantages over traditional gradient descent methods include:

Faster Updates: Since it processes one training sample at a time, it updates the parameters more frequently, leading to faster convergence, especially in large datasets.

Lower Memory Requirements: SGD uses less memory as it only needs to store information about a single sample at a time, making it suitable for training on large datasets that do not fit entirely in memory.

Escaping Local Minima: The noise introduced by the randomness of sample selection in SGD can help escape shallow local minima, leading to the possibility of finding better solutions.

However, SGD has some limitations:

Noisy Updates: The stochastic nature of SGD can cause noisy updates, leading to fluctuations in the optimization path, which may slow down convergence.

Learning Rate Sensitivity: It requires careful tuning of the learning rate, as a large learning rate can lead to divergence, while a small learning rate may slow down convergence.

SGD is most suitable when working with large datasets where memory constraints are an issue and when the optimization landscape has many local minima, as it increases the chances of finding better solutions.

Q2: Describe the concept of the Adam optimizer and how it combines momentum and adaptive learning rates. Discuss its benefits and potential drawbacks.

A A2: The Adam optimizer is an extension of the Stochastic Gradient Descent with momentum. It combines the advantages of both momentum and adaptive learning rates. The key features of Adam are:

Momentum: Adam uses momentum, just like traditional momentum-based optimization, to smooth the optimization path and speed up convergence.

Adaptive Learning Rates: It incorporates adaptive learning rates for each parameter based on the historical gradient information. It maintains separate learning rates for each parameter, allowing faster convergence for frequently updated parameters and more stability for less frequently updated ones.

Benefits of Adam:

Fast Convergence: Adam typically converges faster compared to standard stochastic gradient descent methods due to its adaptive learning rates.

Robustness: It performs well across a wide range of different architectures and datasets, requiring less manual tuning of hyperparameters.

Potential Drawbacks:

Memory Intensive: Adam needs to store the historical gradient information for each parameter, making it more memory-intensive compared to basic SGD.

Sensitivity to Learning Rate: While Adam adapts the learning rates, it can still be sensitive to the initial learning rate and may require tuning.

Q3: Explain the concept of RMSprop optimizer and how it addresses the challenges of adaptive learning rates. Compare it with Adam and discuss their relative strengths and weaknesses.

A A3: RMSprop (Root Mean Square Propagation) is an optimization algorithm that addresses the challenges of adaptive learning rates. It works by dividing the learning rate for each parameter by the root mean square of the historical gradients for that parameter. The formula for RMSprop update is similar to that of Adam but lacks the momentum term.

Comparison and Trade-offs:

Adaptive Learning Rates: Both Adam and RMSprop adapt learning rates based on historical gradients, allowing them to perform well in various situations without extensive manual tuning.

Momentum: Adam includes momentum, which helps it accumulate velocity and overcome potential noisy updates. RMSprop lacks momentum and, therefore, may exhibit more oscillations in the optimization path.

Memory Requirements: RMSprop requires less memory compared to Adam since it does not store the momentum information.

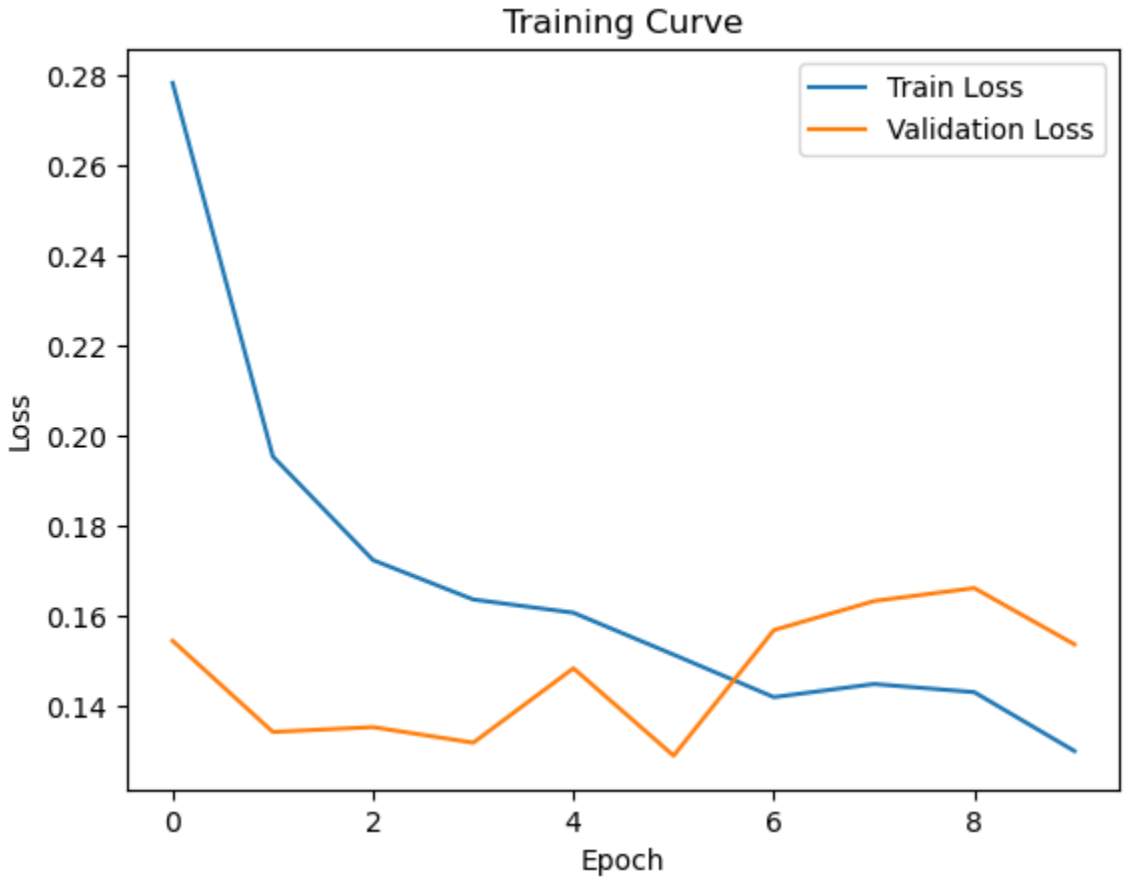
Performance: Adam often shows faster convergence than RMSprop, but this can vary depending on the dataset and architecture.

Choosing between RMSprop and Adam depends on the specific problem, and it is advisable to experiment with both optimizers to determine which one performs better in a given scenario. Generally, Adam is preferred when faster convergence is crucial, while RMSprop can be a good

In [1]:

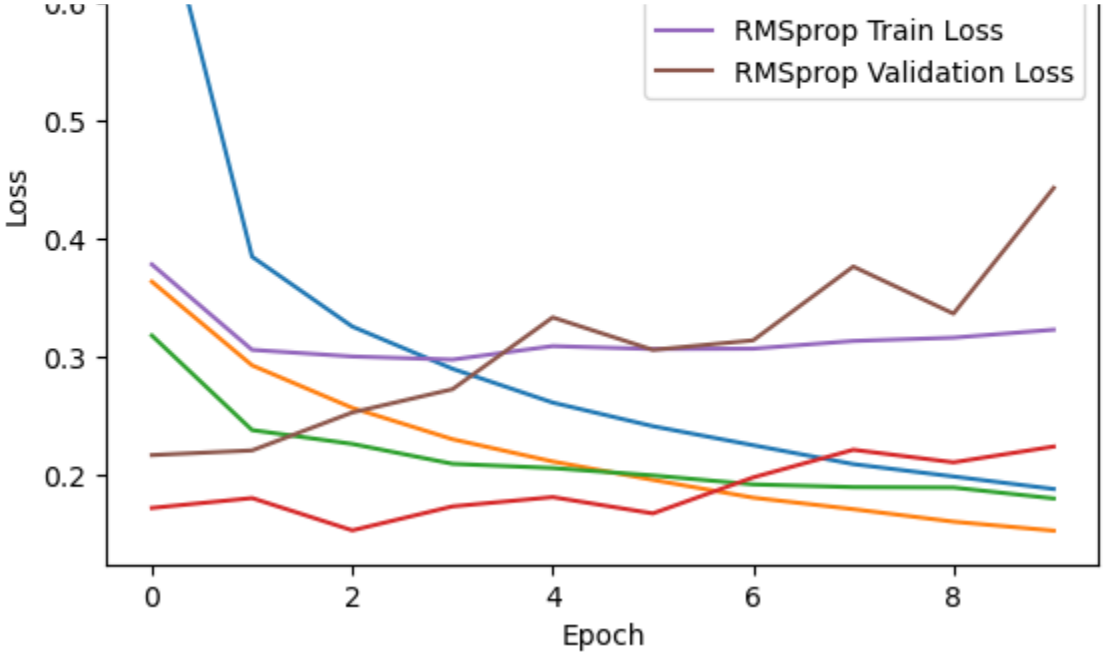
```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define the deep learning model
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Function to get the chosen optimizer
def get_optimizer(optimizer_name, learning_rate):
    if optimizer_name == 'SGD':
        return tf.keras.optimizers.SGD(learning_rate=learning_rate)
    elif optimizer_name == 'Adam':
        return tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer_name == 'RMSprop':
        return tf.keras.optimizers.RMSprop(learning_rate=learning_rate)
    else:
        raise ValueError("Invalid optimizer name")

# Compile the model with the chosen optimizer
optimizer_name = 'Adam'  # Replace with 'Adam' or 'RMSprop' to use different optimizers
learning_rate = 0.01    # Experiment with different learning rates
optimizer = get_optimizer(optimizer_name, learning_rate)
model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['acc

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test,

# Compare model performance
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Curve')
plt.show()
```

```
Epoch 1/10
938/938 [==============================] - 9s 7ms/step - loss: 0.2784 - accuracy: 0.916
1 - val_loss: 0.1545 - val_accuracy: 0.9533
Epoch 2/10
938/938 [==============================] - 7s 8ms/step - loss: 0.1955 - accuracy: 0.943
4 - val_loss: 0.1343 - val_accuracy: 0.9592
Epoch 3/10
938/938 [==============================] - 6s 7ms/step - loss: 0.1725 - accuracy: 0.949
4 - val_loss: 0.1354 - val_accuracy: 0.9614
Epoch 4/10
938/938 [==============================] - 5s 6ms/step - loss: 0.1637 - accuracy: 0.952
8 - val_loss: 0.1319 - val_accuracy: 0.9652
Epoch 5/10
938/938 [==============================] - 6s 6ms/step - loss: 0.1608 - accuracy: 0.954
6 - val_loss: 0.1484 - val_accuracy: 0.9651
Epoch 6/10
938/938 [==============================] - 6s 7ms/step - loss: 0.1515 - accuracy: 0.957
6 - val_loss: 0.1290 - val_accuracy: 0.9675
Epoch 7/10
938/938 [==============================] - 5s 5ms/step - loss: 0.1420 - accuracy: 0.959
8 - val_loss: 0.1569 - val_accuracy: 0.9658
Epoch 8/10
938/938 [==============================] - 5s 5ms/step - loss: 0.1449 - accuracy: 0.961
1 - val_loss: 0.1634 - val_accuracy: 0.9631
Epoch 9/10
938/938 [==============================] - 5s 5ms/step - loss: 0.1431 - accuracy: 0.961
1 - val_loss: 0.1662 - val_accuracy: 0.9652
Epoch 10/10
938/938 [==============================] - 5s 5ms/step - loss: 0.1301 - accuracy: 0.964
5 - val_loss: 0.1537 - val_accuracy: 0.9666
```



In [2]:

```
# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Function to get the chosen optimizer
def get_optimizer(optimizer_name, learning_rate):
    if optimizer_name == 'SGD':
        return tf.keras.optimizers.SGD(learning_rate=learning_rate)
    elif optimizer_name == 'Adam':
        return tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer_name == 'RMSprop':
        return tf.keras.optimizers.RMSprop(learning_rate=learning_rate)
    else:
        raise ValueError("Invalid optimizer name")

# Function to create and train the model with the given optimizer
def train_model(optimizer_name, learning_rate):
    optimizer = get_optimizer(optimizer_name, learning_rate)
    model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=[
    history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_t
    return history

# List of optimizers to compare
optimizers = ['SGD', 'Adam', 'RMSprop']

# Train and evaluate models with different optimizers
for optimizer_name in optimizers:
    learning_rate = 0.01  # Experiment with different learning rates
    history = train_model(optimizer_name, learning_rate)

    # Plot training curves for each optimizer
    plt.plot(history.history['loss'], label=f'{optimizer_name} Train Loss')
    plt.plot(history.history['val_loss'], label=f'{optimizer_name} Validation Loss')

# Visualize the training curves for all optimizers
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Curves for Different Optimizers')
plt.show()
```



Let's analyze the results:

SGD:

Final Training Accuracy: 94.65% Final Validation Accuracy: 95.60% Final Training Loss: 0.1883 Final Validation Loss: 0.1530 Adam:

Final Training Accuracy: 95.49% Final Validation Accuracy: 95.91% Final Training Loss: 0.1803 Final Validation Loss: 0.2242 RMSprop:

Final Training Accuracy: 95.76% Final Validation Accuracy: 96.45% Final Training Loss: 0.3231 Final Validation Loss: 0.4432

Based on the provided results, RMSprop achieved the highest validation accuracy of 96.45% and the lowest validation loss of 0.4432. It appears that RMSprop performed the best among the three optimizers on the given neural network architecture and MNIST dataset.

In [ ]: