Gradient Descent

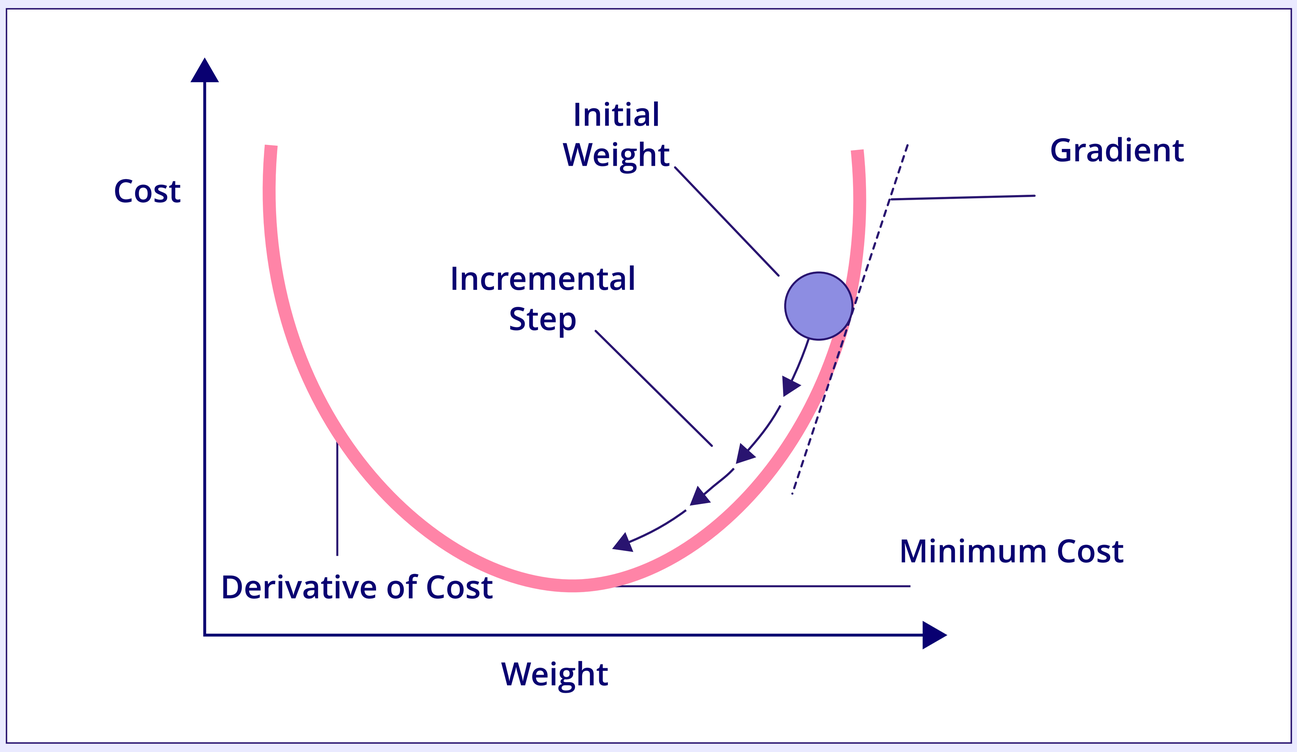
educative

Introduction

Gradient Descent is an optimization algorithm used to minimize the cost function in various machine learning algorithms.

Purpose and Importance:

It iteratively adjusts model parameters to find the minimum of the cost function, crucial for model accuracy and performance.



Applications:

- Linear Regression
- Logistic Regression
- Neural Networks
- Support Vector Machines

Basic Concepts

Gradient:

The gradient is a vector of partial derivatives of the cost function with respect to the model parameters. It indicates the direction of the steepest ascent.

Cost Function:

Measures the error between predicted and actual values. Examples include:

- **Mean Squared Error (MSE)**

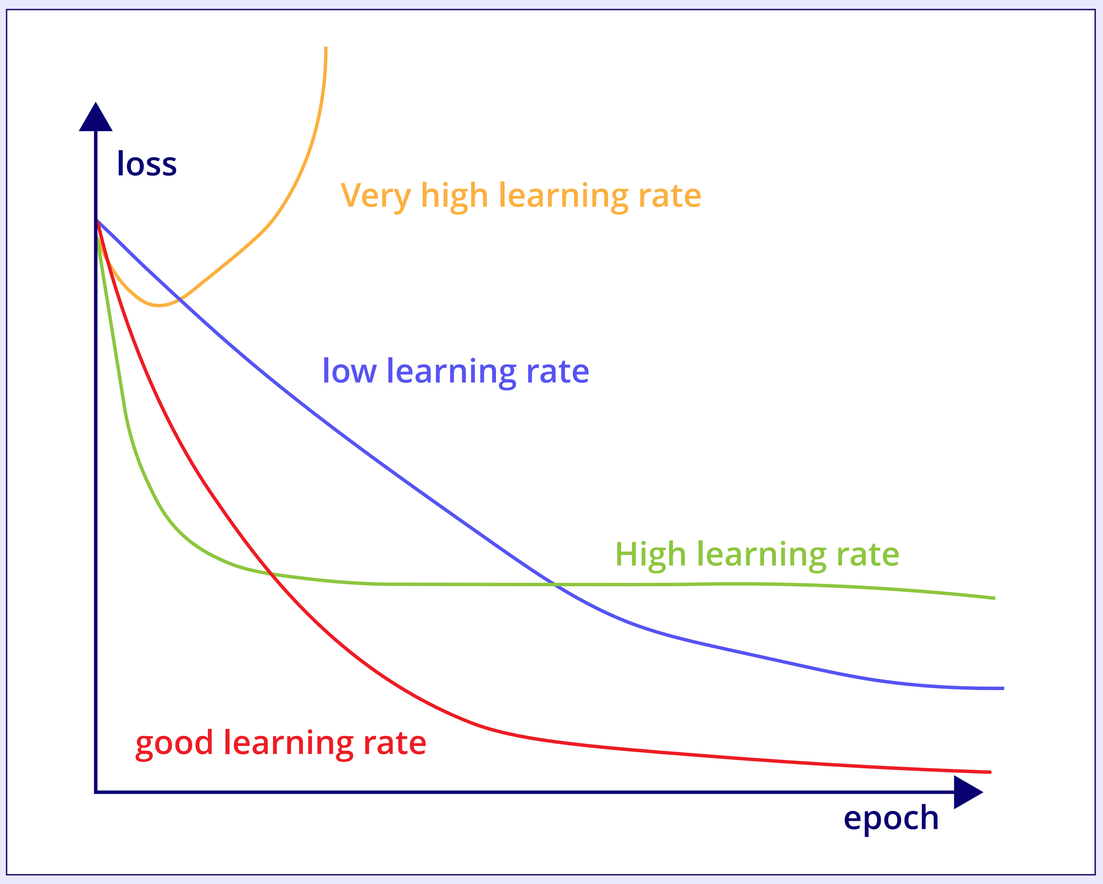
$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- **Cross Entropy**

$$\text{Cross entropy} = \frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Learning Rate (α):

A hyperparameter that controls the step size of each iteration. It significantly impacts convergence speed and model performance. Adaptive methods like Adam and RMSprop are commonly used to optimize learning rates.



Gradient Descent

educative

Gradient Computation:

Automatic differentiation tools like TensorFlow and PyTorch facilitate efficient gradient computation.

```
import tensorflow as tf


# Define a simple model
model = tf.keras.Sequential([tf.keras.layers.Dense(1, input_shape=[1])])


# Define a loss function
loss_fn = tf.keras.losses.MeanSquaredError()


# Compute gradients
with tf.GradientTape() as tape:
    predictions = model(inputs)
    loss = loss_fn(targets, predictions)
gradients = tape.gradient(loss, model.trainable_variables)
```

Types of Gradient Descent

	Batch Gradient Descent	Stochastic Gradient Descent	Mini-Batch Gradient Descent
Description	Uses the entire dataset to compute the gradient.	Uses one training example per iteration.	Uses a subset of the dataset per iteration.
Pros	Accurate and stable updates.	Faster updates, can escape local minima.	Balance between speed and stability.
Cons	Computationally expensive for large datasets.	High variance in updates, can be unstable.	Requires choosing an optimal batch size.
Use Cases	Preferred for smaller datasets or when high precision is required. Often used in linear regression and neural networks when computation power is not a limiting factor.	Ideal for large-scale machine learning tasks, online learning, and scenarios where data is too large to fit in memory. Often used in deep learning models for quick convergence.	Commonly used in deep learning with neural net-works, where it balances the efficiency of SGD with the stability of batch gradient descent. Useful when the dataset is large, but not too large to handle in manageable batches.

Examples:

Batch Gradient Descent

```
for epoch in range(num_epochs):
    gradient = compute_gradient(cost_function, data)
    parameters -= learning_rate * gradient
```

Stochastic Gradient Descent

```
for epoch in range(num_epochs):
    for x, y in data:
        gradient = compute_gradient(cost_function, x, y)
        parameters -= learning_rate * gradient
```

Gradient Descent

educative

Mini-Batch Gradient Descent

```
for epoch in range(num_epochs):
    for mini_batch in get_mini_batches(data, batch_size):
        gradient = compute_gradient(cost_function, mini_batch)
        parameters -= learning_rate * gradient
```

Learning Rate Tuning

Impact on Convergence:

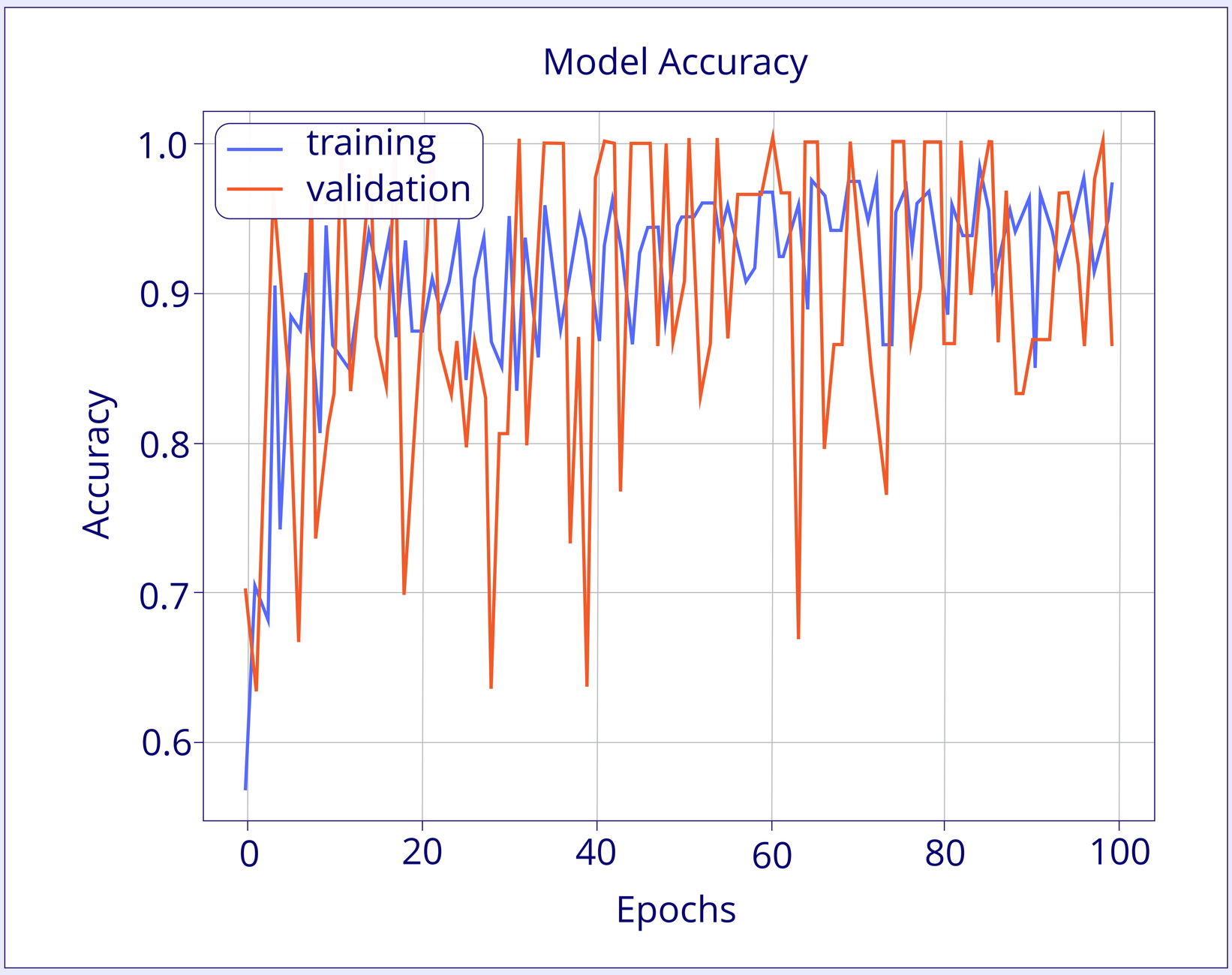
- High learning rate can overshoot minima.
- Low learning rate can lead to slow convergence.

Strategies:

1: Constant Learning Rate

Fixed throughout training.

```
keras.optimizers.SGD(lr=0.1, momentum=0.0, decay=0.0, nesterov=False)
```

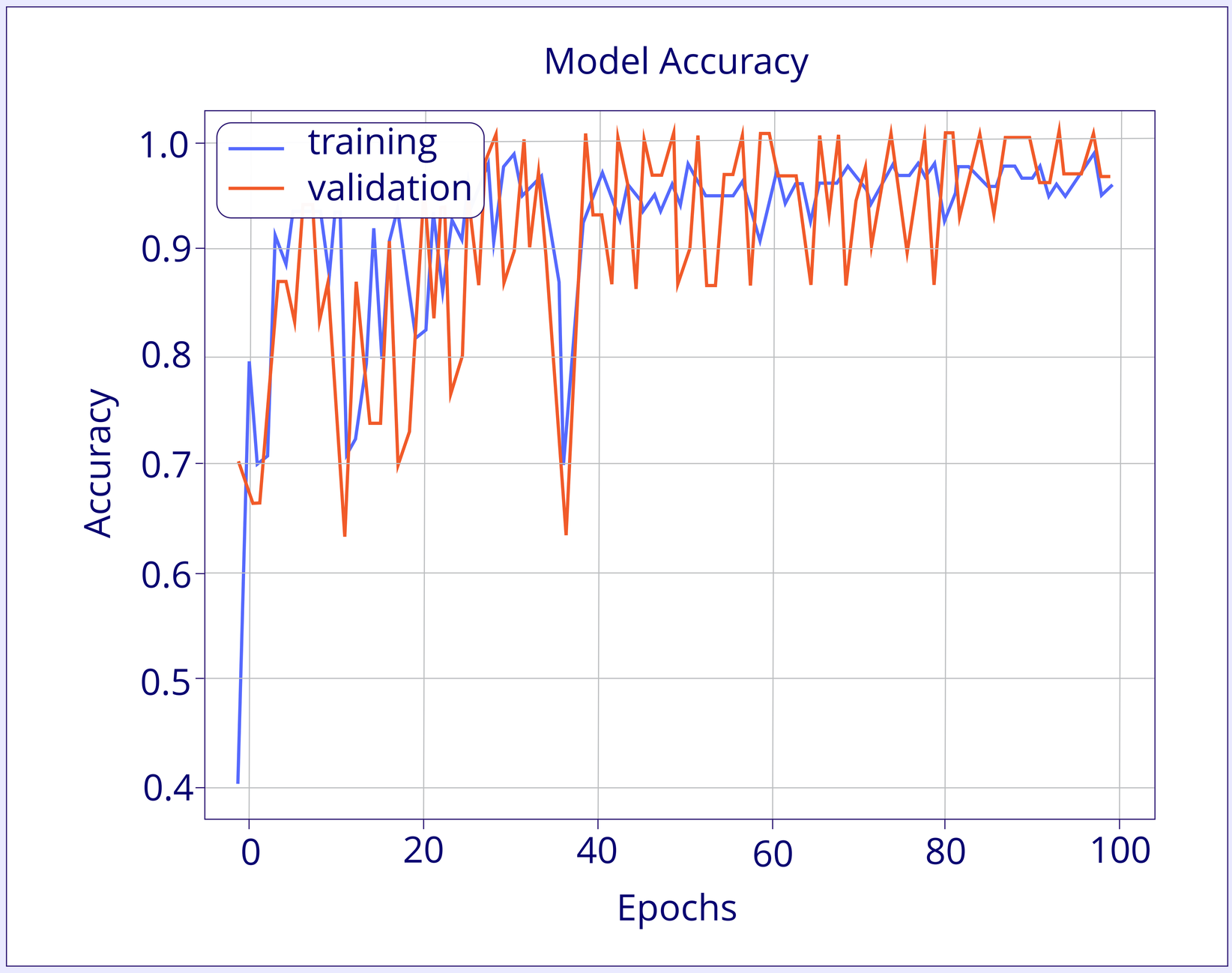


2: Learning Rate Schedules:

Gradually decrease learning rate during training.

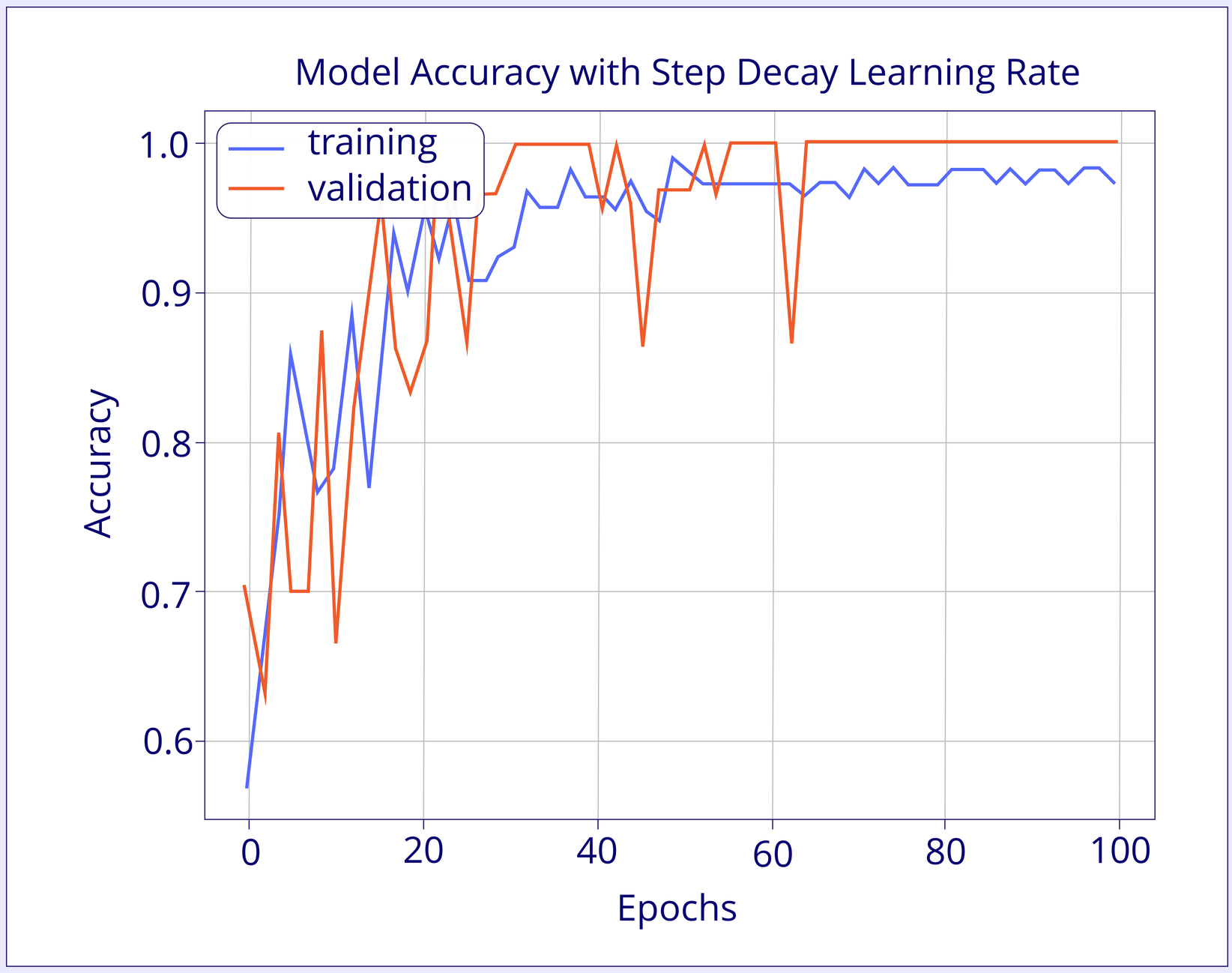
Time-Based Decay

```
# Compile the model with SGD optimizer
learning_rate = 0.05
decay_rate = learning_rate / epochs
momentum = 0.8
optimizer = SGD(lr=learning_rate, momentum=momentum, decay=decay_rate, nesterov=False)
```

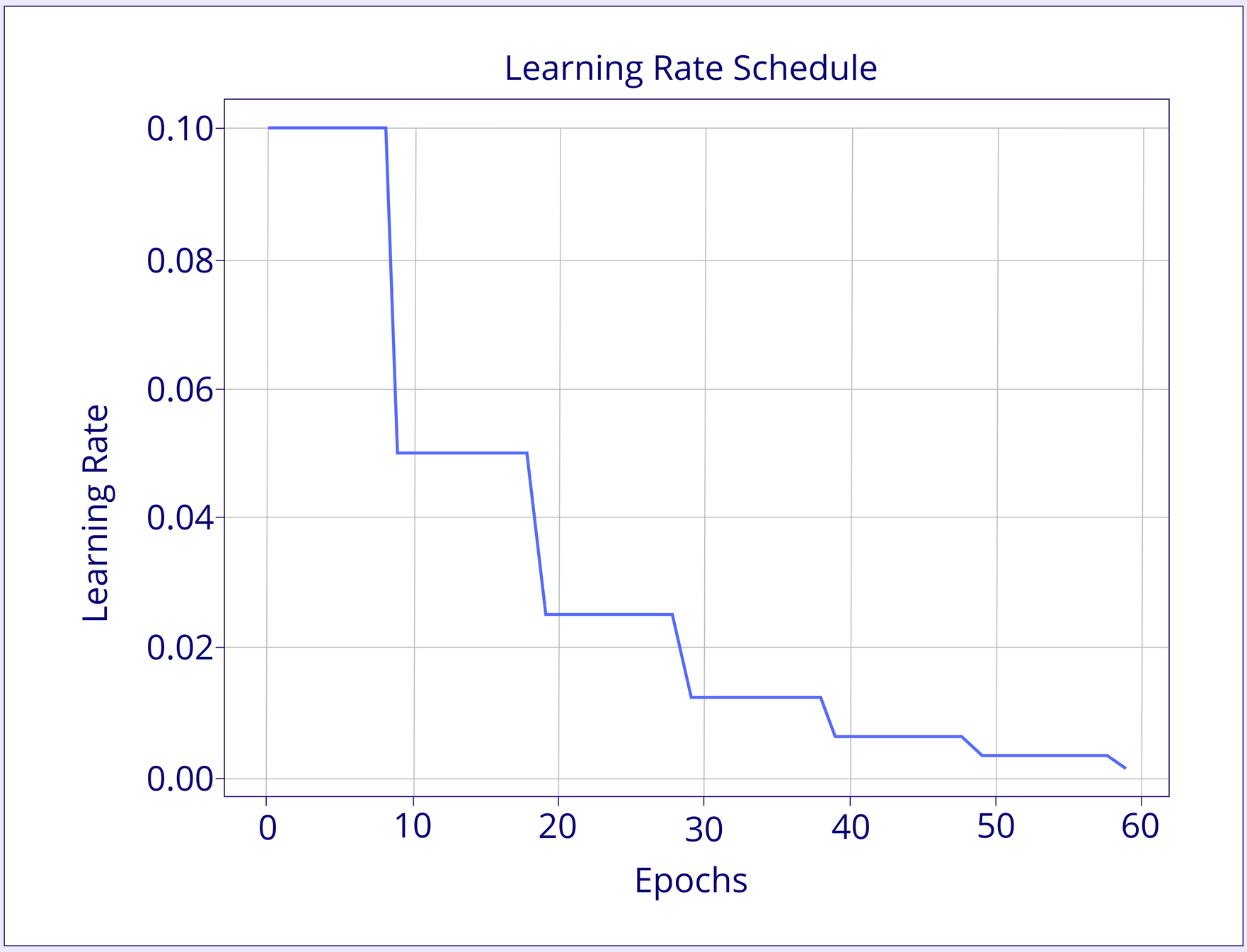


Step-Based Decay

```
# Define step decay function
def step_decay(epoch):
    initial_lrate = 0.1
    drop = 0.5
    epochs_drop = 10.0
    lrate = initial_lrate * math.pow(drop, math.floor((1 + epoch) /   epochs_drop))
    return lrate
```

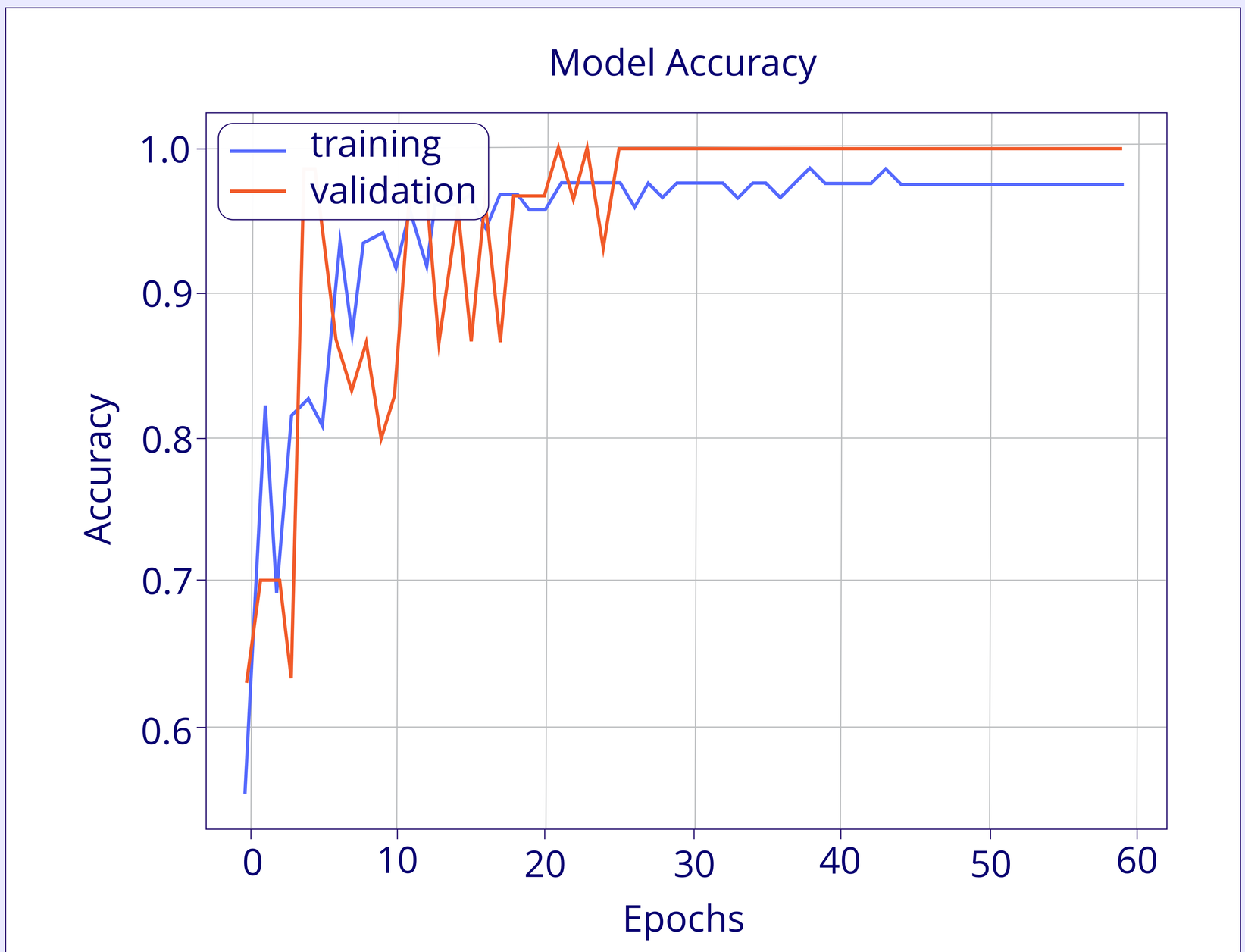


```
# Learning rate scheduler
lrate = LearningRateScheduler(step_decay)
```

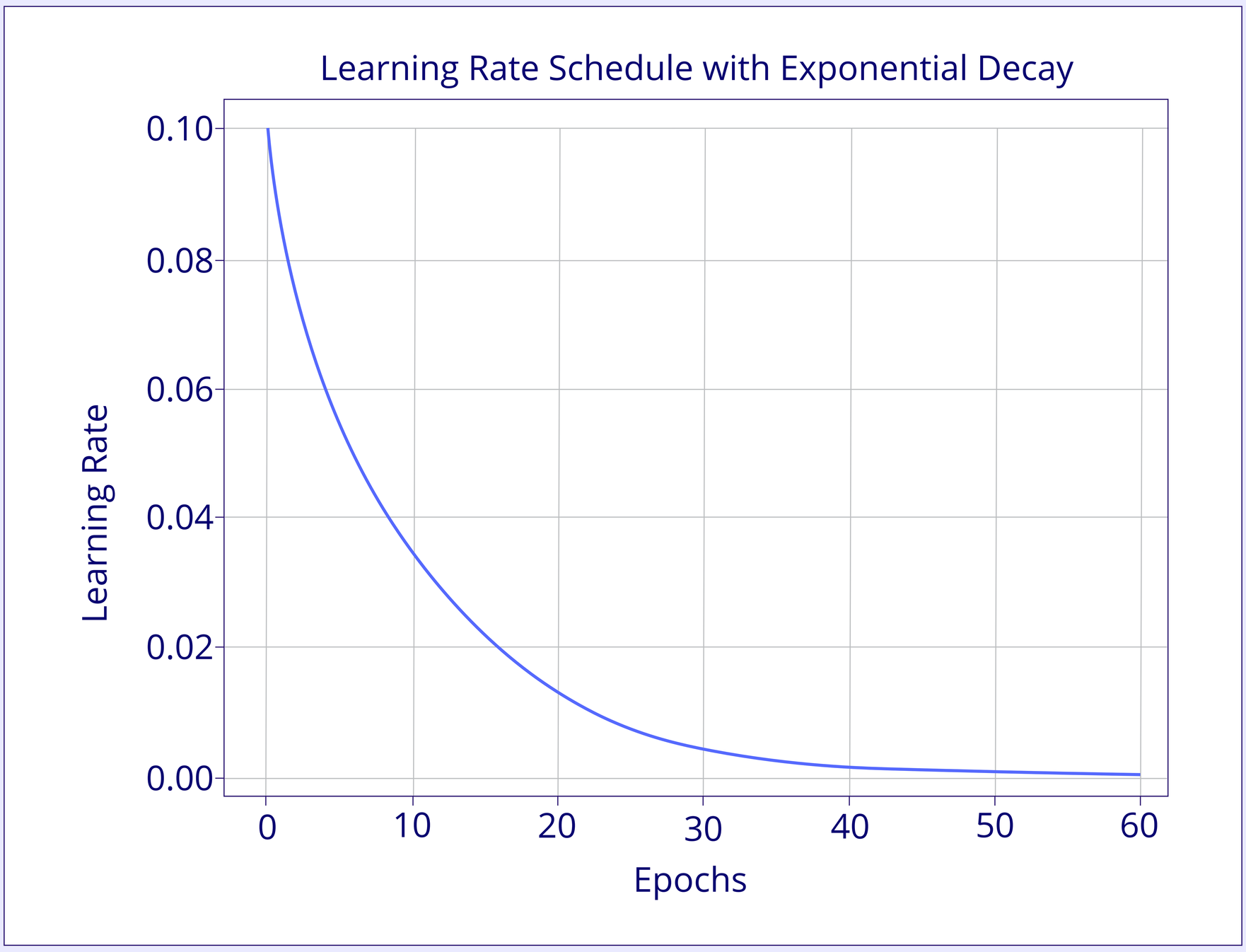


Exponential Decay

```
# Define exponential decay function
def exp_decay(epoch):
    initial_lrate = 0.1
    k = 0.1
    lrate = initial_lrate * math.exp(-k * epoch)
    return lrate
```



```
# Learning rate scheduler
lrate = LearningRateScheduler(exp_decay)
```



3: Adaptive Learning Rates:

Algorithms that adjust the learning rate during training based on the gradient.

Key Algorithms			
	AdaGrad	**RMSprop**	**Adam**
Strengths	Adapts learning rates for each parameter; performs well with sparse data.	Resolves AdaGrad's diminishing learning rate problem by using a moving average of squared gradients.	Combines the benefits of AdaGrad and RMSprop; maintains per-parameter learning rates, updates with both mean and variance of gradients.
Weaknesses	Learning rate continually decays, potentially becoming too small.	Requires tuning of the decay parameter.	More complex with additional hyperparameters (e.g., beta1, beta2).

AdaGrad

```
from keras.optimizers import Adagrad


# Initialize the Adagrad optimizer with a specific learning rate
optimizer = Adagrad(learning_rate=0.01)


# Compile the model with the Adagrad optimizer, categorical crossentropy loss, and
accuracy metric
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


# Train the model using the training data, with validation data provided for evaluation,
# and run for 100 epochs
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100)
```

RMSprop

```
from keras.optimizers import RMSprop


# Initialize the RMSprop optimizer with a specified learning rate
optimizer = RMSprop(learning_rate=0.001)


# Compile the model with the RMSprop optimizer, categorical crossentropy loss, and
accuracy metric
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


# Train the model using the training data, with validation data provided for evaluation,
# and run for 100 epochs
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100)
```

Adam

```
from keras.optimizers import Adam


# Initialize the Adam optimizer with a specified learning rate
optimizer = Adam(learning_rate=0.001)


# Compile the model with the Adam optimizer, categorical crossentropy loss, and
accuracy metric
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


# Train the model using the training data, with validation data provided for evaluation,
# and run for 100 epochs
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100)
```

4: Advanced Techniques

Cyclical Learning Rates, Learning Rate Range Test, Cosine Annealing.

Cyclical Learning Rates (CLR):

Learning rate cyclically varies between a minimum and maximum boundary.
Benefits: Helps escape local minima and can lead to faster convergence.

Implementation Steps:

i. **Initialize Parameters:** Set `base_lr`, `max_lr`, `step_size`, and `mode` (e.g., '`triangular`').
ii. **Calculate Learning Rate:** Use the formula.

```
lr = base_lr + (max_lr - base_lr) * abs((iteration / stepsize) % 2 - 1)
```

iii. **Update Learning Rate:** Implement a callback to adjust the learning rate during training

Code example

```
# Cyclical Learning Rate Callback
class CyclicalLearningRate(Callback):
    def __init__(self, base_lr=0.001, max_lr=0.006, step_size=2000., mode='triangular'):
        super(CyclicalLearningRate, self).__init__()
        self.base_lr = base_lr
        self.max_lr = max_lr
        self.step_size = step_size
        self.mode = mode
        self.iterations = 0.
        self.history = {}


    def clr(self):
        cycle = np.floor(1 + self.iterations / (2 * self.step_size))
        x = np.abs(self.iterations / self.step_size - 2 * cycle + 1)
        if self.mode == 'triangular':
            return self.base_lr + (self.max_lr - self.base_lr) * np.maximum(0, (1 - x))
        else:
            raise ValueError('mode must be "triangular"')
```

Cosine Annealing

The learning rate starts high and decreases following a cosine function.
Benefits: Smoothly reduces the learning rate, which can help in fine-tuning the model toward the end of training.

Implementation Steps:

i. **Initialize Parameters:** Set max_lr, min_lr, and total_epochs.
ii. **Calculate Learning Rate:** Use the formula:

```
lr = min_lr + 0.5 * (max_lr - min_lr) * (1 + np.cos(iteration / total_iterations * np.pi))
```

iii. **Apply Learning Rate Scheduler:** Use a scheduler callback to update the learning rate during training.

Code example

```
# Cosine Annealing Scheduler
def cosine_annealing(epoch, max_lr=0.01, min_lr=0.0001, total_epochs=100):
    return min_lr + 0.5 * (max_lr - min_lr) * (1 + np.cos(np.pi * epoch / total_epochs))


# Usage in a Learning Rate Scheduler Callback
from keras.callbacks import LearningRateScheduler


lr_scheduler = LearningRateScheduler(lambda epoch: cosine_annealing(epoch))
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100, callbacks=
[lr_scheduler])
```

Learning Rate Range Test

Helps find the optimal learning rate by gradually increasing the learning rate and observing the loss.
Benefits: Identifies a good range of learning rates to use for training.

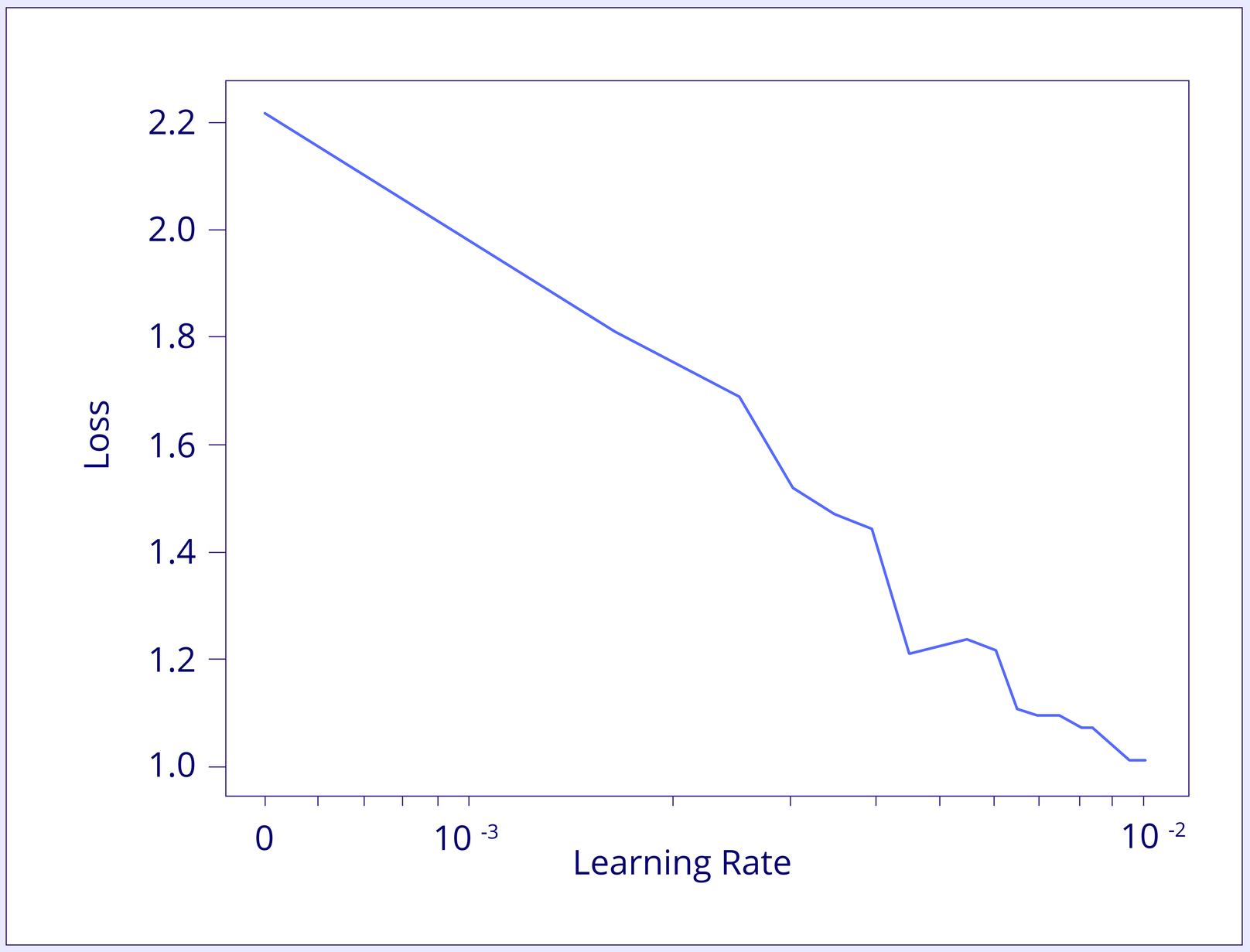
Implementation Steps:

i. Train the model for a few epochs with a learning rate that increases linearly or exponentially.
ii. Plot the learning rate against the loss.
iii. Choose a learning rate that is approximately 10 times lower than the learning rate at which the loss starts to increase.

```
# Learning Rate Finder Callback
class LearningRateFinder(Callback):
    def __init__(self, min_lr=1e-5, max_lr=1e-2, steps_per_epoch=None, epochs=1):
        super(LearningRateFinder, self).__init__()
        self.min_lr = min_lr
        self.max_lr = max_lr
        self.total_steps = steps_per_epoch * epochs
        self.steps_per_epoch = steps_per_epoch
        self.epoch = 0
        self.batch = 0
        self.history = {}


    def on_batch_end(self, batch, logs=None):
        self.batch += 1
        lr = self.min_lr + (self.max_lr - self.min_lr) * (self.batch / self.total_steps)
        K.set_value(self.model.optimizer.lr, lr)
        self.history.setdefault('lr', []).append(lr)
        self.history.setdefault('loss', []).append(logs.get('loss'))
```

Gradient Descent



Convergence and Stopping Criteria

When to Stop

- Change in the cost function is minimal.
- Maximum number of iterations reached.

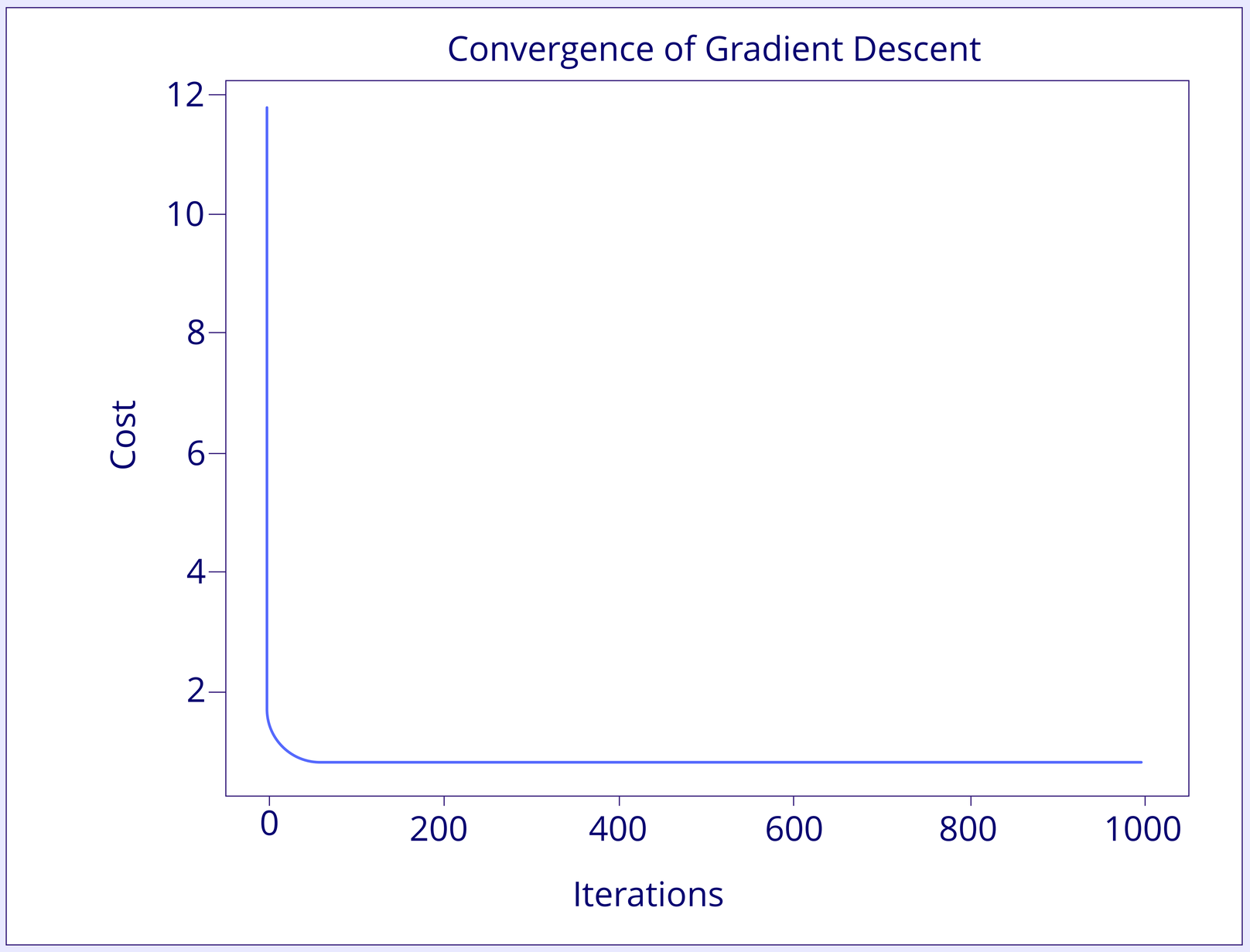
Common Stopping Criteria

- Tolerance level for the change in cost.
- Predefined maximum iterations.

Visualizing Convergence

- Plotting cost function vs. iterations to observe convergence behavior.

```
# Plotting convergence
plt.plot(range(num_iterations), cost_history)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Convergence of Gradient Descent')
plt.show()
```



Common Challenges and Solutions

Local Minima vs. Global Minima

Gradient descent can get stuck in local minima, especially in non-convex functions.

Strategies to Avoid Local Minima:

Random Restarts: Start training with different initial weights.
Momentum: Helps accelerate gradient vectors in the right directions, leading to faster convergence.
Adaptive Learning Rates: Algorithms like Adam, RMSprop dynamically adjust learning rates based on past gradient information, helping to avoid local minima and improve convergence.

Overfitting and Underfitting

Overfitting: The model performs well on training data but poorly on validation data.
Underfitting: The model performs poorly on both training and validation data.1

Solutions:

Regularization: Techniques like L1 and L2 regularization add a penalty for larger weights, which reduces model complexity and helps prevent overfitting.

```
# L1 Regularization (Lasso Regression)
lasso = Lasso(alpha=0.1)  # alpha is the regularization strength
lasso.fit(X_train, y_train)
y_pred_lasso = lasso.predict(X_test)
mse_lasso = mean_squared_error(y_test, y_pred_lasso)


# L2 Regularization (Ridge Regression)
ridge = Ridge(alpha=1.0)  # alpha is the regularization strength
ridge.fit(X_train, y_train)
y_pred_ridge = ridge.predict(X_test)
mse_ridge = mean_squared_error(y_test, y_pred_ridge)
```

Cross-Validation: Use techniques like k-fold cross-validation to better estimate model performance.

```
kf = KFold(n_splits=5, shuffle=True, random_state=0)
```

Dropout: Randomly drops units (along with their connections) from the neural network during training, helping prevent overfitting by making the network less sensitive to specific weights.

```
# Define the neural network model with dropout
model = Sequential([
    Dense(64, activation='relu', input_shape=(20,)),
    Dropout(0.5),  # Dropout layer with 50% dropout rate
    Dense(32, activation='relu'),
    Dropout(0.5),  # Another Dropout layer with 50% dropout rate
    Dense(1, activation='sigmoid')
])
```

Exploding and Vanishing Gradients

Exploding Gradients: Gradients grow exponentially during training, causing model parameters to become unstable.
Solution: Gradient Clipping caps the gradients during backprop

```
# Example of Gradient Clipping in TensorFlow
optimizer = tf.keras.optimizers.Adam(clipvalue=1.0)
```

Vanishing Gradients: Gradients shrink during training, making it difficult for the model to learn.
Solution: Use ReLU activation or similar, and careful weight initialization.

```
# Example of Using ReLU in TensorFlow
model.add(tf.keras.layers.Dense(64, activation='relu'))
```

Practical Tips

Feature Scaling

Standardizing (mean of 0 and standard deviation of 1) or normalizing (rescaling to [0, 1]) features can improve convergence and overall model performance.

Data Preprocessing

Clean and preprocess data to improve model performance.

Techniques:

Handling Missing Values: Impute missing data using mean, median, or mode, or remove incomplete rows/columns.

```
# Impute missing values with column mean
df.fillna(df.mean(), inplace=True)
```

Encoding Categorical Variables: Convert categorical data into numerical formats using techniques like one-hot encoding or label encoding.

```
# Convert categorical variables to numerical format
df = pd.get_dummies(df, columns=['category'])
```

Feature Scaling: Normalize or standardize features to ensure they are on a similar scale, which can help gradient descent converge faster.

```
# Standardize features to have mean=0 and variance=1
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df)
```

Outlier Detection: Identify and handle outliers to prevent them from skewing the model's performance.

```
# Filter outliers outside of bounds
df = df[(df['col'] > lower_bound) & (df['col'] < upper_bound)]
```

Data Splitting: Divide data into training, validation, and test sets to evaluate model performance accurately.

```
# Split data into training and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Initialization of Parameters

Use techniques like Xavier or He initialization for neural networks.

```
# Example of He Initialization in TensorFlow
initializer = tf.keras.initializers.HeNormal()
layer = tf.keras.layers.Dense(units=64, kernel_initializer=initializer)
```

Monitoring Convergence

Plot Cost Function Against Iterations:

Track the cost function over iterations using the validation set to ensure proper convergence.

Early Stopping:

Halt training if validation performance starts to degrade, preventing overfitting.

Advanced Tips

Leverage pretrained models and transfer learning for faster convergence and better performance.

```
# Example of Transfer Learning in TensorFlow
base_model = tf.keras.applications.VGG16(input_shape=(224, 224, 3), include_top=False,
weights='imagenet')
model = tf.keras.Sequential([
    base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(10, activation='softmax')
])
```