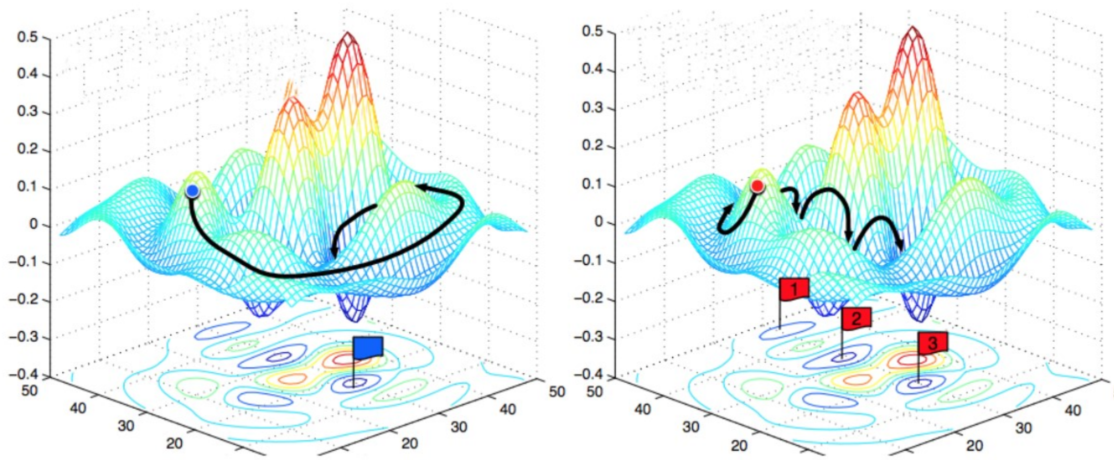
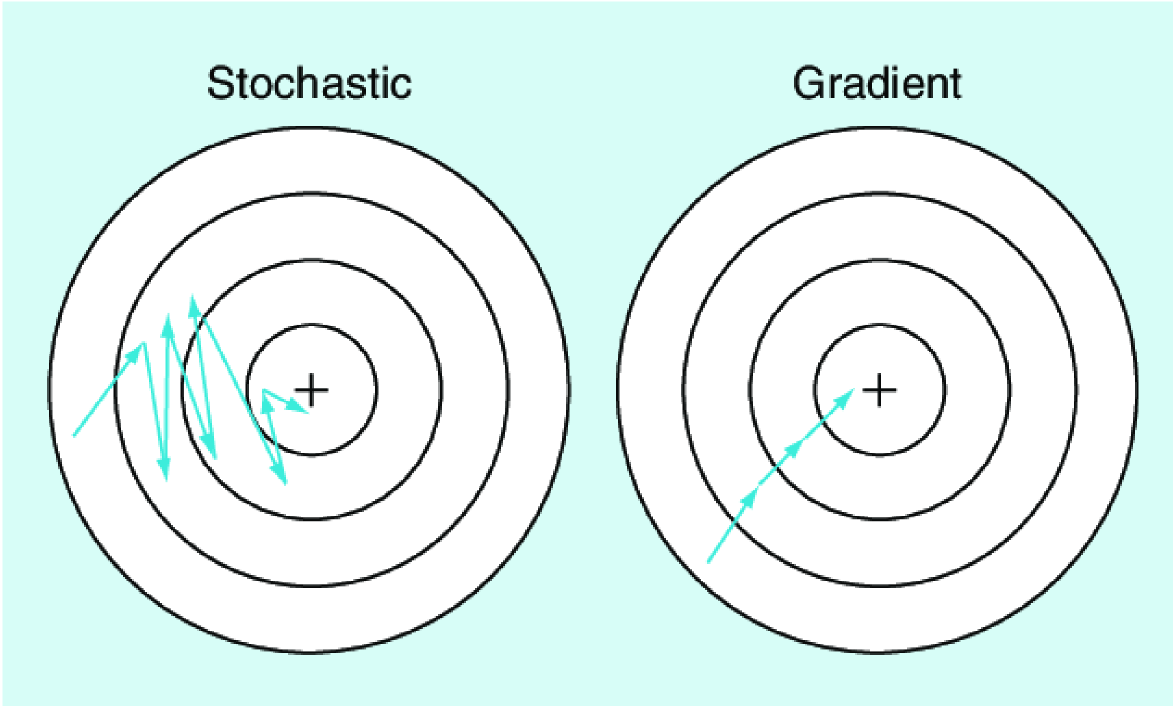
What is Stochastic Gradient Descent (SGD) ?

Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used in machine learning and deep learning for **finding the optimal parameters of a model**. It is a variation of gradient descent that updates the model's parameters using the gradients computed on a randomly selected subset of the training dataset at each iteration. This makes SGD more **computationally efficient** compared to batch gradient descent while still maintaining a **good convergence rate**".



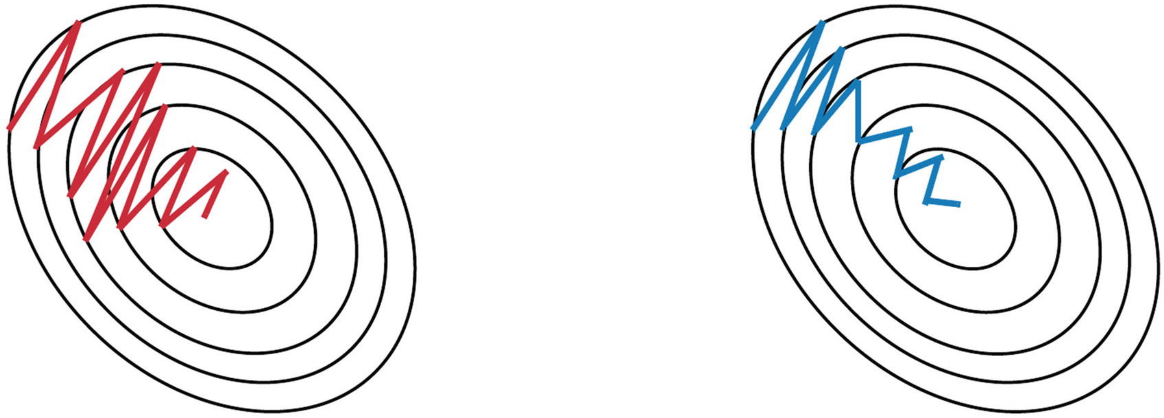
Stochastic gradient descent (SGD) is an **iterative optimization algorithm used to minimize a function**. It is a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in high-dimensional optimization problems this reduces the very high computational burden, achieving faster iterations in exchange for a lower convergence rate.

- SGD works by iteratively updating the model parameters in the direction of the negative gradient of the cost function. However, instead of using the entire training dataset to calculate the gradient, SGD uses a single training example or a small batch of training examples. This makes **SGD much faster than batch gradient descent, but it can also make it less stable**.



- The main advantage of SGD is that it is **very computationally efficient**. This makes it a **good choice for large datasets**. Additionally, SGD can be **used to train models with non-convex cost functions**, which can be difficult to train with other optimization algorithms.
- However, SGD can also be **less stable than other optimization algorithms**. This is because the gradient of the cost function can be very noisy, especially for small batches. This noise can make it difficult for the algorithm to converge to a minimum.
- To address this issue, we can use a technique called **momentum**. Momentum helps to smooth out the noise in the gradient, which can help the algorithm **converge more quickly**.

In general, SGD is a **good choice for large datasets**. However, if the dataset is small or noisy, then other optimization algorithms may be a better choice.



Stochastic Gradient Descent **withhout** Momentum

Stochastic Gradient Descent **with** Momentum

Here are some of the benefits and drawbacks of SGD:

Benefits:

- Computationally efficient
- Can be used to train models with non-convex cost functions
- Can be used to train models on large datasets

Drawbacks:

- Can be less stable than other optimization algorithms
- Can be sensitive to noise

Here are some examples of when SGD might be used:

- Training a linear regression model on a large dataset
- Training a neural network on a large dataset
- Training a model on a dataset with a lot of noise

Step by Step Process

Let's explore the steps involved in stochastic gradient descent in more detail:

1. **Initialize Parameters**: Start by initializing the model's parameters, such as weights and biases, with random values. These parameters will be iteratively updated during the training process.

2. **Define the Cost Function**: Choose an appropriate cost function that measures the discrepancy between the predicted values of the model and the actual values in the training dataset. The choice of cost function depends on the specific problem at hand.

3. **Choose a Learning Rate**: Select a learning rate, which determines the step size taken in the direction of the gradients during parameter updates. The learning rate is a

hyperparameter that needs to be carefully chosen. A large learning rate may cause overshooting, while a small learning rate can result in slow convergence.

4. **Repeat until Convergence**:

a. **Randomly Shuffle the Training Dataset**: Shuffle the training dataset randomly. This step is crucial to introduce randomness in the training process and avoid potential biases due to the order of the instances.

b. **For each Training Instance**:

```
  i. **Randomly Select a Training Instance**: Select a single training instance randomly from the shuffled dataset.

  ii. **Forward Propagation**: Pass the selected training instance through the model to obtain the predicted output. Apply the necessary activation functions and use the current parameter values.

  iii. **Compute Loss**: Calculate the cost function using the predicted output and the actual target for the selected training instance. This step quantifies the model's performance on that specific instance.

  iv. **Backpropagation**: Perform backpropagation to compute the gradients of the cost function with respect to each parameter. Backpropagation involves propagating the error gradients backward through the layers of the model using the chain rule of calculus.

  v. **Update Parameters**: Update the parameters of the model using the gradients computed from the selected training instance. The update rule for each parameter is given by:

     `parameter = parameter - learning_rate * gradient`

     Here, the gradient represents the derivative of the cost function with respect to the parameter being updated.
```

c. **Repeat Steps 4b for Each Training Instance**: Iterate over all the training instances in the shuffled dataset, applying forward propagation, computing loss, backpropagation, and parameter updates for each instance.

5. **Repeat Steps 4a-4c**: Repeat the process of random shuffling the dataset and iterating over the instances until a stopping criterion is met. The stopping criterion can be a maximum number of iterations or a threshold on the improvement of the cost function.

6. **Evaluate Model**: Once the training process is complete, evaluate the trained model's performance on a separate validation or test dataset. This step gives an estimate of how well the model is likely to generalize to unseen data.

Stochastic gradient descent has several advantages, including computational efficiency due to updates based on subsets of instances, **the ability to escape local minima**, and the potential for **faster convergence**. However, it may exhibit more fluctuations in the optimization process due to the noisy gradients computed on individual instances. To strike a balance between computational efficiency and convergence stability, other variations of gradient descent, such

In [1]:
```
# code

import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
```

In [2]:
```
from sklearn.datasets import load_diabetes

# Load the diabetes dataset
X, y = load_diabetes(return_X_y=True)
```

In [3]:
```
print(X.shape)
print(y.shape)
```

```
(442, 10)
(442,)
```

In [4]:
```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_stat
```

In [5]:
```
# Import the required library/dependency
from sklearn.linear_model import LinearRegression

# Create an instance of the LinearRegression class
reg = LinearRegression()

# Train the model using the training data
reg.fit(X_train, y_train)
```

Out[5]:
```
▾ LinearRegression
LinearRegression()
```

In [6]:
```
# Print the coefficients of the linear regression model
print(reg.coef_)

# Print the intercept of the linear regression model
print(reg.intercept_)
```

```
[  -9.15865318 -205.45432163  516.69374454  340.61999905 -895.5520019
  561.22067904  153.89310954  126.73139688  861.12700152   52.42112238]
151.88331005254167
```

In [7]:
```
# Predict using the regression model
y_pred = reg.predict(X_test)

# Calculate the R2 score
r2_score(y_test, y_pred)
```

Out[7]: 0.4399338661568968

In [8]:
```
X_train.shape
```

Out[8]: (353, 10)

In [9]:
```
class SGDRegressor:
    """A class representing a SGD Regressor."""

    def __init__(self, learning_rate=0.01, epochs=100):
        """
        Initialize the SGDRegressor.

        Parameters:
        - learning_rate (float): The learning rate for gradient descent (defau
        - epochs (int): The number of epochs for training (default: 100).
        """
        self.coef_ = None
        self.intercept_ = None
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self, X_train, y_train):
        """
        Fit the SGDRegressor to the training data.

        Parameters:
        - X_train (ndarray): The input training data.
        - y_train (ndarray): The target training data.
        """
        # Initialize coefficients
        self.intercept_ = 0
        self.coef_ = np.ones(X_train.shape[1])

        for i in range(self.epochs):
            for j in range(X_train.shape[0]):
                idx = np.random.randint(0, X_train.shape[0])

                y_hat = np.dot(X_train[idx], self.coef_) + self.intercept_

                # Calculate derivative of intercept
                intercept_der = -2 * (y_train[idx] - y_hat)
                self.intercept_ = self.intercept_ - (self.lr * intercept_der)

                # Calculate derivative of coefficients
                coef_der = -2 * np.dot((y_train[idx] - y_hat), X_train[idx])
                self.coef_ = self.coef_ - (self.lr * coef_der)

        print(self.intercept_, self.coef_)

    def predict(self, X_test):
        """
        Predict the output for the test data.

        Parameters:
        - X_test (ndarray): The input test data.

        Returns:
        - ndarray: The predicted output for the test data.
        """
```

```
        return np.dot(X_test, self.coef_) + self.intercept_
```

Explanation

The code in the focal cell defines a class called SGDRegressor, which represents a stochastic gradient descent (SGD) regressor. This class allows you to perform linear regression using stochastic gradient descent as the optimization algorithm.

Here's a line-by-line explanation of the code:

- The code starts by defining the SGDRegressor class. It has two parameters: learning_rate (the learning rate for gradient descent, with a default value of 0.01) and epochs (the number of epochs for training, with a default value of 100).

1. The **init** method is the constructor of the class. It initializes the instance variables coef_ and intercept_ as None, and assigns the provided learning_rate and epochs values to the corresponding instance variables.
2. The fit method is used to train the SGDRegressor on the provided training data. It takes X_train (the input training data) and y_train (the target training data) as parameters.
3. Inside the fit method, the coefficients (coef_) and intercept (intercept_) are initialized. The intercept is set to 0, and the coefficients are set to an array of ones with the same shape as the number of features in the input data.
4. The training process starts with two nested loops. The outer loop iterates epochs number of times, and the inner loop iterates over each sample in the training data.
5. In each iteration of the inner loop, a random index (idx) is generated to select a random training sample.
6. The predicted output (y_hat) for the selected sample is calculated using the dot product of the input data (X_train[idx]) and the coefficients (self.coef_), plus the intercept (self.intercept_).
7. The derivative of the intercept (intercept_der) is calculated as -2 * (y_train[idx] - y_hat). This derivative is used to update the intercept by subtracting the learning rate (self.lr) multiplied by the derivative.
8. The derivative of the coefficients (coef_der) is calculated as -2 * np.dot((y_train[idx] - y_hat), X_train[idx]). This derivative is used to update the coefficients by subtracting the learning rate multiplied by the derivative.
9. After the training process is complete, the final intercept and coefficients are printed using print(self.intercept_, self.coef_).
10. The predict method takes the test data (X_test) as input and returns the predicted output for the test data. It calculates the predicted output using the dot product of the test data and the coefficients, plus the intercept.

In [10]:
```
sgd = SGDRegressor(learning_rate=0.01,epochs=40)
```

In [11]:
```
import time

start = time.time()
sgd.fit(X_train,y_train)
print("The time taken is",time.time() - start)
```

```
155.025148437934 [  56.25772805  -40.56378569  318.57811154  230.43923781   2
7.63190136
  -12.63851034 -164.27077296  130.91101253  294.70051101  128.7553447 ]
The time taken is 0.18108773231506348
```

In [12]:
```
y_pred = sgd.predict(X_test)
```

In [13]:
```
r2_score(y_test,y_pred)
```

Out[13]:
```
0.4202489926600458
```

Sklearn

In [14]:
```
from sklearn.linear_model import SGDRegressor
```

In [15]:
```
reg = SGDRegressor(max_iter=100,learning_rate='constant',eta0=0.01)
```

In [16]:
```
reg.fit(X_train,y_train)
```

```
C:\Users\user\anaconda3\lib\site-packages\sklearn\linear_model\_stochastic_gr
adient.py:1548: ConvergenceWarning: Maximum number of iteration reached befor
e convergence. Consider increasing max_iter to improve the fit.
  warnings.warn(
```

Out[16]:
```
▾ SGDRegressor
SGDRegressor(learning_rate='constant', max_iter=100)
```

In [17]:
```
# Predict using the regression model
y_pred = reg.predict(X_test)
```

In [18]:
```
# Calculate the R2 score
r2_score(y_test,y_pred)
```

Out[18]:
```
0.43300205052916463
```

Animation

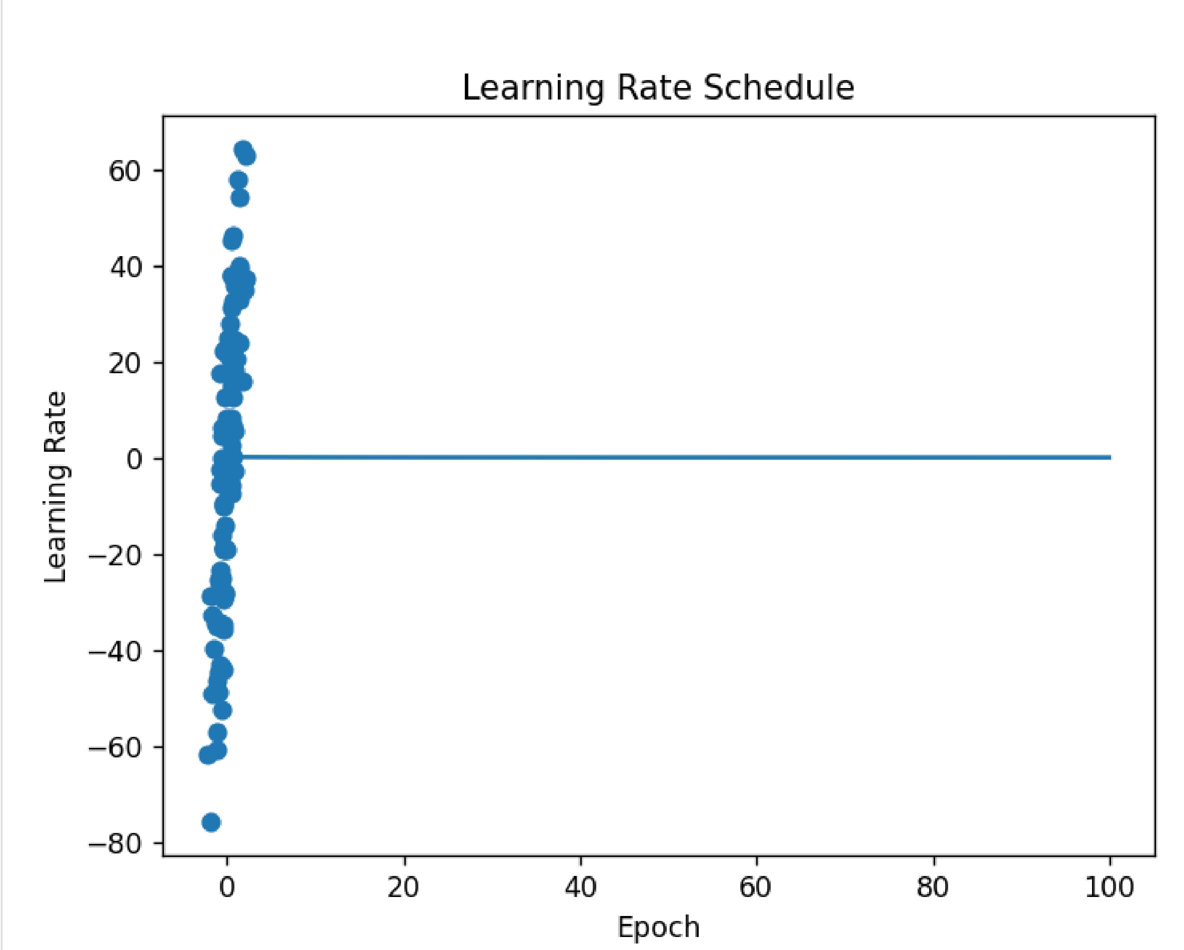
In [19]:
```
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression

import numpy as np
import matplotlib.pyplot as plt
```

In [20]:
```
%matplotlib notebook
from matplotlib.animation import FuncAnimation
import matplotlib.animation as animation
```

In [21]:
```
X,y = make_regression(n_samples=100, n_features=1, n_informative=1, n_targets=
```

In [22]:
```
plt.scatter(X,y)
```

Figure 1



Out[22]: <matplotlib.collections.PathCollection at 0x200873c6af0>

In [23]:
```
# Fitting linear regression model
lr = LinearRegression()
lr.fit(X, y)

# Printing the coefficients and intercept
print(lr.coef_)
print(lr.intercept_)
```

```
[27.82809103]
-2.29474455867698
```

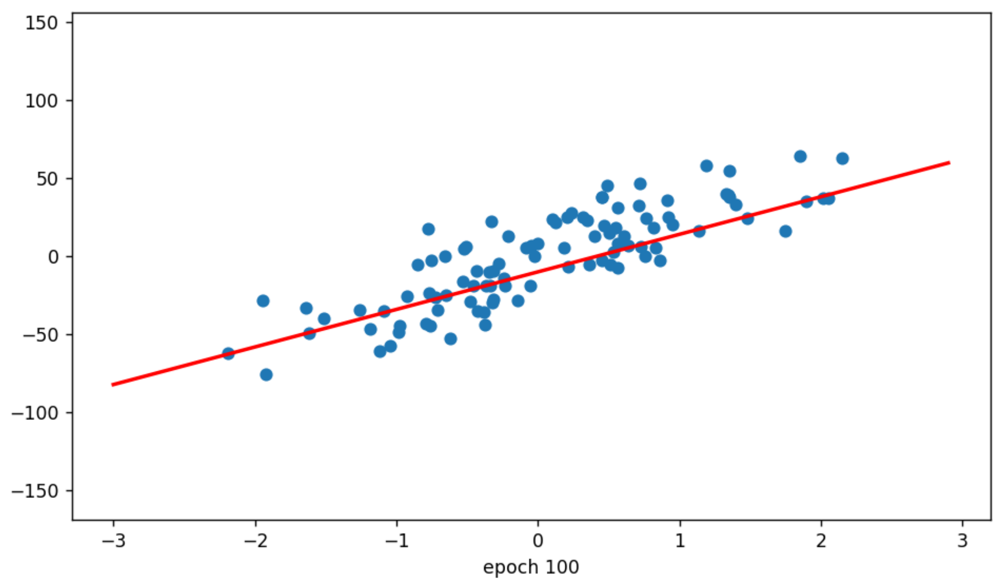
In [24]:
```
import time
import numpy as np

b = 150
m = -127.82
all_b = []
all_m = []
all_cost = []
all_lr = []

epochs = 1

start = time.time()

t0, t1 = 5, 50

def learning_rate(t: int) -> float:
    """Calculate the learning rate."""
    return t0 / (t + t1)

for i in range(epochs):
    for j in range(X.shape[0]):
        lr = learning_rate(i * X.shape[0] + j)
        idx = np.random.randint(X.shape[0], size=1)

        slope_b = -2 * (y[idx] - (m * X[idx]) - b)
        slope_m = -2 * (y[idx] - (m * X[idx]) - b) * X[idx]
        cost = (y[idx] - m * X[idx] - b) ** 2

        b = b - (lr * slope_b)
        m = m - (lr * slope_m)
        all_b.append(b)
        all_m.append(m)
        all_cost.append(cost)
        all_lr.append(lr)

print("Total time taken:", time.time() - start)
```

```
Total time taken: 0.008001565933227539
```

In [25]:
```
len(all_cost)
```

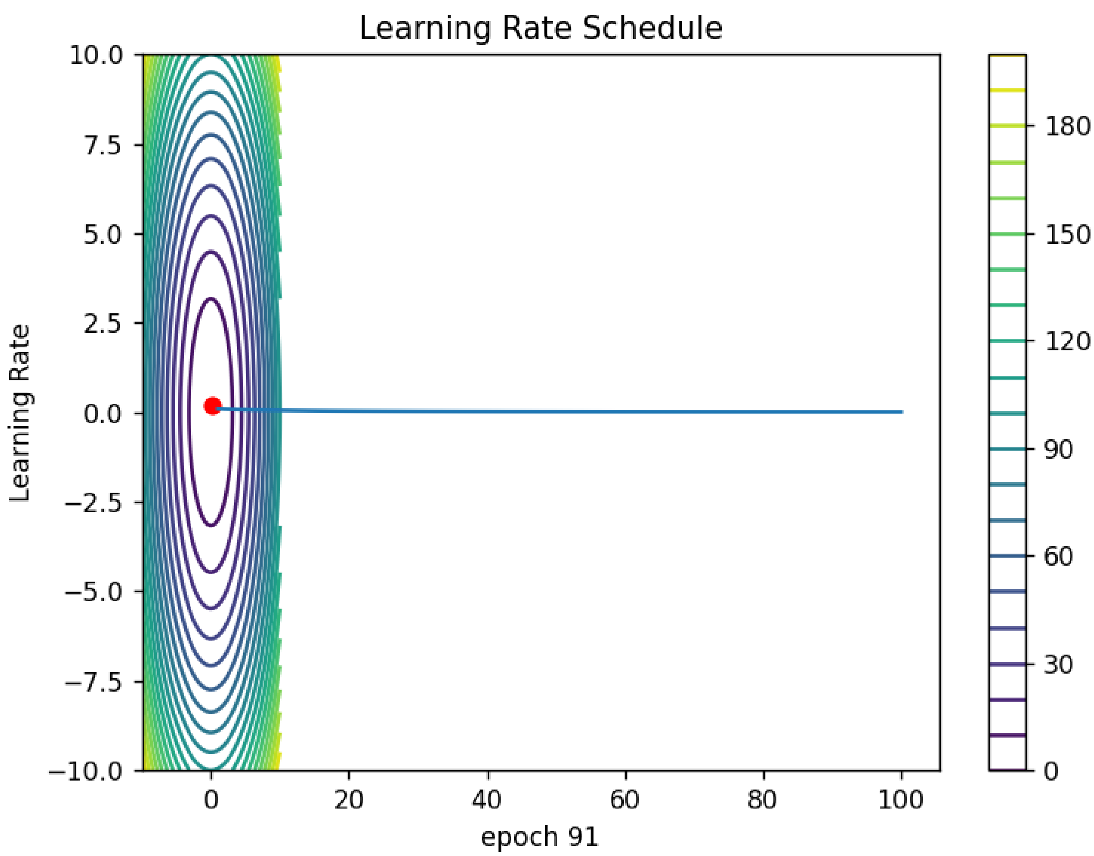
Out[25]:
```
100
```

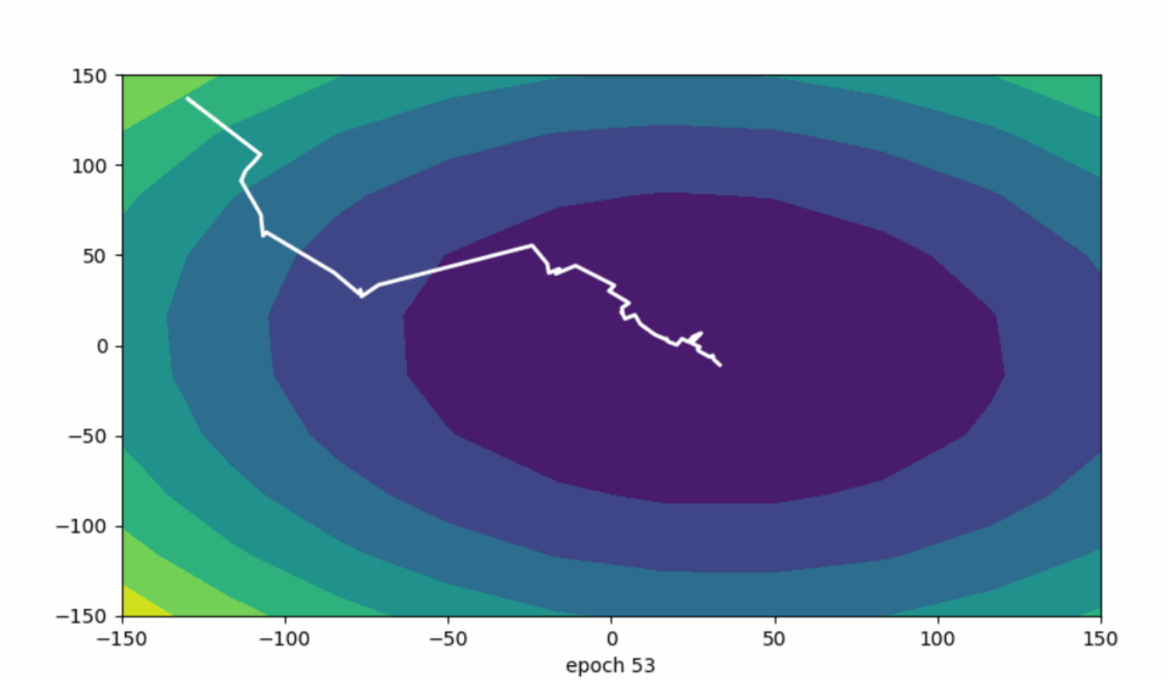
In [26]:
```
fig, ax = plt.subplots(figsize=(9,5))
#fig.set_tight_layout(True)

x_i = np.arange(-3, 3, 0.1)
y_i = x_i*(-27) -150
ax.scatter(X, y)
line, = ax.plot(x_i, x_i*50 - 4, 'r-', linewidth=2)

def update(i):
    label = 'epoch {0}'.format(i + 1)
    line.set_ydata(x_i*all_m[i] + all_b[i])
    ax.set_xlabel(label)
    # return line, ax

anim = FuncAnimation(fig, update, frames=100, interval=5)

f = r"stochastic_animation_line_plot.gif"
writergif = animation.PillowWriter(fps=2)
anim.save(f, writer=writergif)
```



In [27]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Define the cost function
def cost_function(x, y):
    return x**2 + y**2

# Define the gradient of the cost function
def gradient(x, y):
    return 2*x, 2*y

# Define the stochastic gradient descent algorithm
def stochastic_gradient_descent(learning_rate, num_iterations):
    # Initialize the parameters and the trajectory list
    x = 6
    y = 6
    trajectory = [(x, y)]

    for i in range(num_iterations):
        # Compute the gradient at the current point
        grad_x, grad_y = gradient(x, y)

        # Update the parameters using stochastic gradient descent
        x -= learning_rate * grad_x
        y -= learning_rate * grad_y

        # Append the new point to the trajectory
        trajectory.append((x, y))

    return trajectory

# Create the contour plot
x_vals = np.linspace(-10, 10, 100)
y_vals = np.linspace(-10, 10, 100)
X, Y = np.meshgrid(x_vals, y_vals)
Z = cost_function(X, Y)

fig, ax = plt.subplots()
contour = ax.contour(X, Y, Z, levels=20)

# Initialize the scatter plot for the trajectory
scatter = ax.scatter([], [], color='red')

# Update function for the animation
def update(frame):
    x, y = trajectory[frame]
    scatter.set_offsets([(x, y)])
    return scatter,

# Run stochastic gradient descent
learning_rate = 0.1
num_iterations = 50
trajectory = stochastic_gradient_descent(learning_rate, num_iterations)

# Create the animation
animation = FuncAnimation(fig, update, frames=num_iterations+1, interval=200,
```

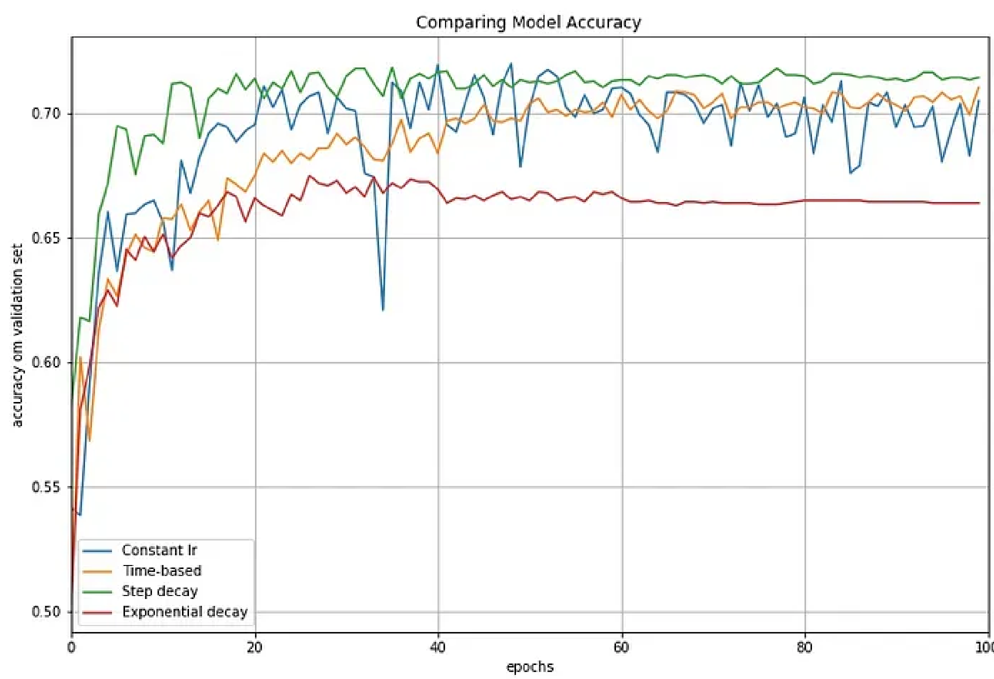
```
# Display the animation
plt.xlabel('x')
plt.ylabel('y')
plt.title('Stochastic Gradient Descent')
plt.colorbar(contour)
plt.tight_layout()
plt.show()
```





Learning schedules in SGD

In Stochastic Gradient Descent (SGD), a learning schedule, also known as a learning rate schedule or learning rate decay, is a technique that adjusts the learning rate during training to improve the convergence and performance of the model. The learning rate determines the step size taken in the direction of the gradients during parameter updates.



Here are a few commonly used learning schedules in SGD:

1. **Fixed Learning Rate**: In this simple approach, the learning rate remains constant throughout the training process. It is set to a fixed value, typically determined through hyperparameter tuning. However, a fixed learning rate may not be ideal for all scenarios, as it may lead to slow convergence or overshooting.
2. **Time-based Decay**: This learning schedule reduces the learning rate over time by a fixed factor at predetermined intervals or epochs. The formula for time-based decay is:

   `learning_rate = initial_learning_rate / (1 + decay_rate * epoch)`

   Here, `initial_learning_rate` is the starting learning rate, `decay_rate` controls the rate of decay, and `epoch` is the current epoch number. As the epochs progress, the learning rate decreases, allowing smaller steps for fine-tuning the model parameters.

3. **Step Decay**: Step decay reduces the learning rate by a fixed factor at specific milestones or steps during training. The formula for step decay is:

   `learning_rate = initial_learning_rate * decay_factor^floor(epoch / step_size)`

   Here, `initial_learning_rate` is the initial learning rate, `decay_factor` is the factor by which the learning rate is reduced, `epoch` is the current epoch number, and `step_size` is the number of epochs after which the learning rate is reduced. The learning rate is reduced after every `step_size` epochs.

4. **Exponential Decay**: Exponential decay reduces the learning rate exponentially over time. The formula for exponential decay is:

   `learning_rate = initial_learning_rate * decay_rate^epoch`

   Here, `initial_learning_rate` is the initial learning rate, `decay_rate` is a value between 0 and 1 that controls the decay rate, and `epoch` is the current epoch number. The learning rate decreases exponentially with each epoch.

5. **Piecewise Constant Decay**: Piecewise constant decay allows you to define specific learning rates for different epochs or ranges of epochs. It is often used to decrease the learning rate more aggressively at the beginning of training and then reduce it more slowly later. The learning rate is manually adjusted based on the desired schedule.

These are just a few examples of learning schedules used in SGD. The choice of learning schedule depends on the problem at hand, the characteristics of the dataset, and empirical experimentation to find the best settings. It is important to note that learning rate schedules should be carefully tuned to achieve the desired balance between convergence speed and avoiding overshooting or getting stuck in local minima.

In [28]:
```
import numpy as np

# Define hyperparameters
initial_learning_rate = 0.1
decay_rate = 0.1
epochs = 100

# Initialize theta parameter
theta = 0.0

# Define your gradient function
def compute_gradient():
    # Replace this with your actual gradient computation
    # Make sure it returns the computed gradient
    gradient = 0.0  # Compute the gradient here
    return gradient

# Training Loop
for epoch in range(epochs):
    # Compute learning rate for the current epoch
    learning_rate = initial_learning_rate / (1 + decay_rate * epoch)

    # Compute the gradient
    gradient = compute_gradient()

    # Perform SGD update using the current learning rate
    theta -= learning_rate * gradient

    # Print the learning rate for each epoch
    print(f"Epoch {epoch+1}: Learning Rate = {learning_rate}")
```

```
Epoch 1: Learning Rate = 0.1
Epoch 2: Learning Rate = 0.09090909090909091
Epoch 3: Learning Rate = 0.08333333333333334
Epoch 4: Learning Rate = 0.07692307692307693
Epoch 5: Learning Rate = 0.07142857142857144
Epoch 6: Learning Rate = 0.06666666666666667
Epoch 7: Learning Rate = 0.0625
Epoch 8: Learning Rate = 0.058823529411764705
Epoch 9: Learning Rate = 0.05555555555555556
Epoch 10: Learning Rate = 0.052631578947368425
Epoch 11: Learning Rate = 0.05
Epoch 12: Learning Rate = 0.047619047619047616
Epoch 13: Learning Rate = 0.045454545454545456
Epoch 14: Learning Rate = 0.04347826086956522
Epoch 15: Learning Rate = 0.041666666666666664
Epoch 16: Learning Rate = 0.04
Epoch 17: Learning Rate = 0.038461538461538464
Epoch 18: Learning Rate = 0.037037037037037035
Epoch 19: Learning Rate = 0.03571428571428572
Epoch 20: Learning Rate = 0.034482758620689655
Epoch 21: Learning Rate = 0.03333333333333333
Epoch 22: Learning Rate = 0.03225806451612903
Epoch 23: Learning Rate = 0.03125
Epoch 24: Learning Rate = 0.030303030303030304
Epoch 25: Learning Rate = 0.029411764705882353
Epoch 26: Learning Rate = 0.028571428571428574
Epoch 27: Learning Rate = 0.02777777777777778
Epoch 28: Learning Rate = 0.02702702702702703
Epoch 29: Learning Rate = 0.02631578947368421
Epoch 30: Learning Rate = 0.02564102564102564
Epoch 31: Learning Rate = 0.025
Epoch 32: Learning Rate = 0.02439024390243903
Epoch 33: Learning Rate = 0.023809523809523808
Epoch 34: Learning Rate = 0.02325581395348837
Epoch 35: Learning Rate = 0.022727272727272728
Epoch 36: Learning Rate = 0.022222222222222223
Epoch 37: Learning Rate = 0.02173913043478261
Epoch 38: Learning Rate = 0.02127659574468085
Epoch 39: Learning Rate = 0.020833333333333332
Epoch 40: Learning Rate = 0.02040816326530612
Epoch 41: Learning Rate = 0.02
Epoch 42: Learning Rate = 0.0196078431372549
Epoch 43: Learning Rate = 0.019230769230769232
Epoch 44: Learning Rate = 0.018867924528301889
Epoch 45: Learning Rate = 0.018518518518518517
Epoch 46: Learning Rate = 0.018181818181818184
Epoch 47: Learning Rate = 0.017857142857142856
Epoch 48: Learning Rate = 0.017543859649122806
Epoch 49: Learning Rate = 0.017241379310344827
Epoch 50: Learning Rate = 0.01694915254237288
Epoch 51: Learning Rate = 0.016666666666666666
Epoch 52: Learning Rate = 0.01639344262295082
Epoch 53: Learning Rate = 0.016129032258064516
Epoch 54: Learning Rate = 0.015873015873015872
Epoch 55: Learning Rate = 0.015625
Epoch 56: Learning Rate = 0.015384615384615385
Epoch 57: Learning Rate = 0.015151515151515152
```

```
Epoch 58: Learning Rate = 0.014925373134328358
Epoch 59: Learning Rate = 0.014705882352941176
Epoch 60: Learning Rate = 0.014492753623188406
Epoch 61: Learning Rate = 0.014285714285714287
Epoch 62: Learning Rate = 0.014084507042253521
Epoch 63: Learning Rate = 0.013888888888888889
Epoch 64: Learning Rate = 0.0136986301369863
Epoch 65: Learning Rate = 0.013513513513513514
Epoch 66: Learning Rate = 0.013333333333333334
Epoch 67: Learning Rate = 0.013157894736842105
Epoch 68: Learning Rate = 0.012987012987012988
Epoch 69: Learning Rate = 0.01282051282051282
Epoch 70: Learning Rate = 0.012658227848101266
Epoch 71: Learning Rate = 0.0125
Epoch 72: Learning Rate = 0.012345679012345678
Epoch 73: Learning Rate = 0.012195121951219514
Epoch 74: Learning Rate = 0.012048192771084336
Epoch 75: Learning Rate = 0.011904761904761904
Epoch 76: Learning Rate = 0.011764705882352941
Epoch 77: Learning Rate = 0.011627906976744184
Epoch 78: Learning Rate = 0.01149425287356322
Epoch 79: Learning Rate = 0.011363636363636364
Epoch 80: Learning Rate = 0.011235955056179775
Epoch 81: Learning Rate = 0.011111111111111112
Epoch 82: Learning Rate = 0.01098901098901099
Epoch 83: Learning Rate = 0.010869565217391304
Epoch 84: Learning Rate = 0.01075268817204301
Epoch 85: Learning Rate = 0.010638297872340425
Epoch 86: Learning Rate = 0.010526315789473684
Epoch 87: Learning Rate = 0.010416666666666668
Epoch 88: Learning Rate = 0.010309278350515464
Epoch 89: Learning Rate = 0.01020408163265306
Epoch 90: Learning Rate = 0.010101010101010102
Epoch 91: Learning Rate = 0.01
Epoch 92: Learning Rate = 0.009900990099009901
Epoch 93: Learning Rate = 0.00980392156862745
Epoch 94: Learning Rate = 0.009708737864077669
Epoch 95: Learning Rate = 0.009615384615384616
Epoch 96: Learning Rate = 0.009523809523809525
Epoch 97: Learning Rate = 0.009433962264150943
Epoch 98: Learning Rate = 0.009345794392523364
Epoch 99: Learning Rate = 0.009259259259259259
Epoch 100: Learning Rate = 0.009174311926605505
```

In [30]:

```
import numpy as np
import matplotlib.pyplot as plt

# Define hyperparameters
initial_learning_rate = 0.1
decay_rate = 0.1
epochs = 100

# Initialize theta parameter
theta = 0.0

# Define your gradient function
def compute_gradient():
    # Replace this with your actual gradient computation
    # Make sure it returns the computed gradient
    gradient = 0.0  # Compute the gradient here
    return gradient

# Lists to store learning rates and epoch numbers
learning_rates = []
epoch_numbers = []

# Training loop
for epoch in range(epochs):
    # Compute learning rate for the current epoch
    learning_rate = initial_learning_rate / (1 + decay_rate * epoch)

    # Compute the gradient
    gradient = compute_gradient()

    # Perform SGD update using the current learning rate
    theta -= learning_rate * gradient

    # Append learning rate and epoch number to the lists
    learning_rates.append(learning_rate)
    epoch_numbers.append(epoch + 1)

# Plotting the learning rate
plt.plot(epoch_numbers, learning_rates)
plt.xlabel('Epoch')
plt.ylabel('Learning Rate')
plt.title('Learning Rate Schedule')
plt.show()
```

In [ ]: