Stochastic Batch Gradient Descent Exercise

1 Importing Libraries

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

2 Polynomial Loss Function

```
1 def polynomial_loss(theta, X, y):
2     m = len(y)
3     h = np.dot(X, theta)
4     loss = np.sum((h - y) ** 2) / m
5     return loss
```

This loss function calculates the loss using the polynomial loss function. It takes three parameters: `theta` (the current parameters), `X` (the input features), and `y` (the target values). It calculates the hypothesis function `h` by taking the dot product of `X` and `theta`, then computes the squared difference between `h` and `y`, sums it up, and divides by m to get the average loss.

3 Stochastic Batch Gradient Descent

The `stochastic_batch_gradient_descent` function performs stochastic batch gradient descent. It takes five parameters: X (input features), y (target values), batch_size (the number of samples per batch), learning_rate (the learning rate), and num_epochs (the number of training epochs).

```
1 def stochastic_batch_gradient_descent(X, y, batch_size,
      learning_rate, num_epochs):
```

First, initialize the parameters `theta` with zero and create an empty list `theta_history` to store the intermediate theta values. Calculate the number of batches based on the total number of training examples (‘m‘) and the batch size.

```
    m, n = X.shape
    theta = np.zeros(n)
    theta_history = [theta.copy()]
    num_batches = m // batch_size
```

The function iterates over the epochs and within each epoch, it iterates over the batches, which involves the following steps:

1. Select the samples for the current batch using slicing.
2. Compute the hypothesis function `h_batch`, the error `error_batch`, and the gradient using batch operations.
3. Update and save the parameters `theta` using the learning rate and the gradient.
4. Return the final result.

```
    for epoch in range(num_epochs):
        for batch in range(num_batches):
            start_index = batch * batch_size
            end_index = start_index + batch_size
            X_batch = X[start_index:end_index, :]
            y_batch = y[start_index:end_index]

            h_batch = np.dot(X_batch, theta)
            error_batch = h_batch - y_batch
            gradient = np.dot(X_batch.T, error_batch) / batch_size

            theta = theta - learning_rate * gradient
            theta_history.append(theta.copy())

            loss = polynomial_loss(theta, X, y)

    return theta, theta_history
```

4 Main function

Now, let's generate synthetic data: 100 random 2-dimensional points for X and the target values y based on a linear relationship with some random noise.

```
np.random.seed(0)
X = np.random.rand(100, 2)
y = 2 * X[:, 0] + 3 * X[:, 1] + 0.01*np.random.randn(100)
```

Next, set the values for the batch size, learning rate, and number of epochs.

```
batch_size = ?
learning_rate = ?
num_epochs = ?
```

Now you can call the method we wrote in the previous section to perform SGD on the data you just created.

5 Visualize the SGD process

Use the following function to visualize the training process that you saved from the training method.

```
def plot_loss_contour(X, y, theta_history, theta_range= (-10, 10)
    ):
    theta0_vals = np.linspace(theta_range[0], theta_range[1], 100)
    theta1_vals = np.linspace(theta_range[0], theta_range[1], 100)
    theta0_mesh, theta1_mesh = np.meshgrid(theta0_vals,
        theta1_vals)

    loss_vals = np.zeros_like(theta0_mesh)
    for i in range(len(theta0_vals)):
        for j in range(len(theta1_vals)):
            theta = np.array([theta0_mesh[i, j], theta1_mesh[i, j
                ]])
            loss_vals[i, j] = polynomial_loss(theta, X, y)

    plt.figure(figsize=(8, 6))
    plt.contour(theta0_mesh, theta1_mesh, loss_vals, levels=20,
        cmap='jet')

    for i in range(len(theta_history) - 1):
        plt.plot(theta_history[i][0], theta_history[i][1], 'bo-',
            alpha=0.4, linewidth=0.5, markersize=2)
    plt.plot(theta_history[-1][0], theta_history[-1][1], 'ro-',
        alpha=0.8, linewidth=0.5, markersize=2)

    plt.xlabel('theta0')
    plt.ylabel('theta1')
    plt.title('Contour of Loss Function')
    plt.colorbar()
    plt.show()
```

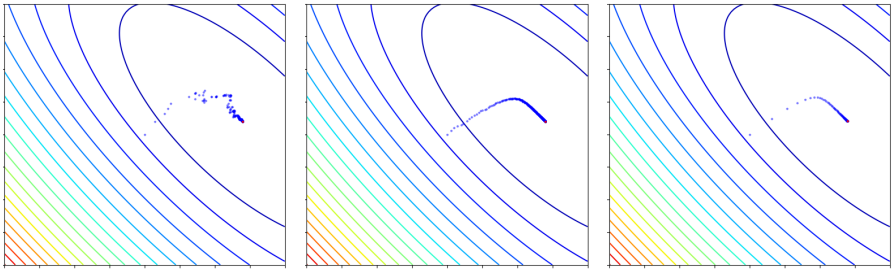
6 Main Task: Parameter Tuning

6.1 Low Noise

If you have successfully completed all the previous sections, you should be able to observe graphs similar to the ones shown below. The data used for generating these graphs is as follows:

```
1 np.random.seed(0)
2 X = np.random.rand(100, 2)
3 y = 7 * X[:, 0] + 1 * X[:, 1] + 0.01*np.random.randn(100)
```

For this section, we will fix the `num_epochs` to be 100. Your task is to tune the `learning_rate` and `batch_size` parameters to match the graphs presented below. **Report your final parameter values at the end of the session.**



Discuss the following questions with your teammates and report your answer at the end of the session.

1. What effect does the batch size control? In other words, what happens if you choose a batch size that is too big or too small?

2. What effect does the learning rate control?

6.2 High Noise

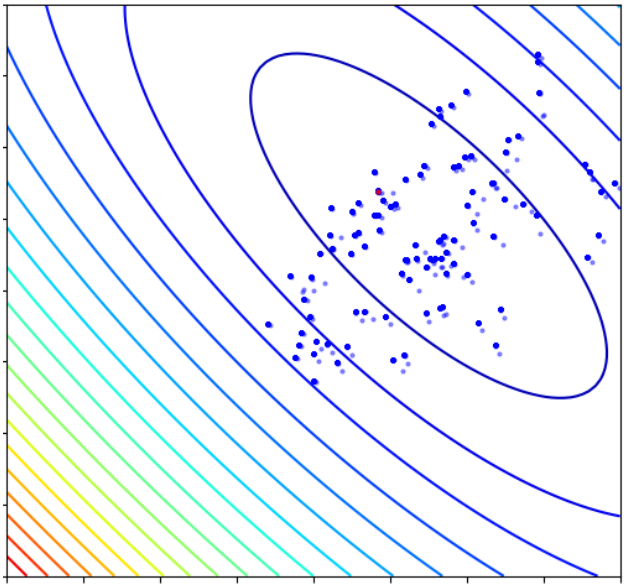
Now, let's consider a scenario where our target contains significantly more (1000x) noise compared to the previous section. The code snippet provided below generates the data:

```
1 np.random.seed(0)
2 X = np.random.rand(100, 2)
3 y = 7 * X[:, 0] + 1 * X[:, 1] + 10*np.random.randn(100)
```

The initial parameter values are set as follows:

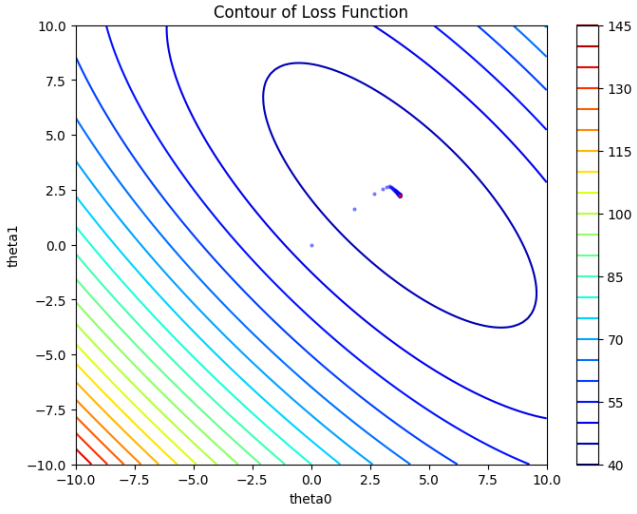
```
1 batch_size = 1
2 learning_rate = 0.3
3 num_epochs = 100
```

The graph corresponding to these parameters appears as follows:



It is evident that the model is not converging effectively in this case. What would you change to make it run better?

Checkpoint: If you tune the hyper parameter well, you should be able to see the figure below.



7 Advanced Task: More Optimizers!

Please modify your code to implement one or more of the following learning methods, other than Stochastic Gradient Descent (SGD). Try your optimizer on both low and high noise input dataset.

- **Momentum SGD**: An extension of SGD that accelerates convergence by adding a fraction of the previous update vector to the current update vector, helping to overcome oscillations and speed up learning.
- **Adam**: An adaptive learning rate optimization algorithm that computes individual learning rates for different parameters based on estimates of first and second moments of the gradients, resulting in faster convergence and reduced sensitivity to hyperparameter tuning.
- **Newton Method**: An optimization technique that uses second-order derivative information (the Hessian matrix) to find the optimal parameters, offering faster convergence than first-order methods but requiring more computation.

8 Optional Reading: Hyperparameter Tuning Strategy

The SGD quick start guide

Newcomers to stochastic gradient descent often find all of these design choices daunting, and it's useful to have simple rules of thumb to get going. We recommend the following:

1. Pick as large a minibatch size as you can given your computer's RAM.
2. Set your momentum parameter to either 0 or 0.9. Your call!
3. Find the largest constant stepsize such that SGD doesn't diverge. This takes some trial and error, but you only need to be accurate to within a factor of 10 here.
4. Run SGD with this constant stepsize until the empirical risk plateaus.
5. Reduce the stepsize by a constant factor (say, 10)
6. Repeat steps 4 and 5 until you converge.

While this approach may not be the most optimal in all cases, it's a great starting point and is good enough for probably 90% of applications we've encountered.

[Recht, Hardt, book 2021]