Synchronous Stocastic Gradient Descent Exercise

1 A Guide to Python Multiprocessing and Parallel Programming

`pool.map` is a method provided by the `multiprocessing.Pool` class in Python's `multiprocessing` module. It allows you to apply a given function to each element of an iterable in parallel across multiple worker processes.

Here's a detailed introduction on how to use `pool.map`:

1. Import the necessary modules:

```
from multiprocessing import Pool
```

2. Define the function that you want to apply in parallel. This function should take a single argument (the input element) and return the result of the computation for that element. Let's call this function `my_function` for demonstration purposes:

```
def my_function(input_element):
    # Perform computation on the input element
    result = ...
    return result
```

3. Create an instance of the `Pool` class, specifying the desired number of worker processes. This number can be based on the available CPU cores or a custom value depending on your requirements. In this example, we'll use 4 worker processes:

```
num_workers = 4
with Pool(processes=num_workers) as pool:
    ...
```

4. Prepare the input data as an iterable. This could be a list, tuple, or any iterable object. Let's call this iterable `input_data`:

```
input_data = [...]
```

5. Use `pool.map` to apply the function to each element of the input data in parallel. Pass the function and the input data as arguments to `pool.map`. The result will be a list of the computed results in the same order as the input data:

   ```
       results = pool.map(my_function, input_data)
   ```

6. Process the results as needed. The `results` list will contain the computed results for each element in the input data.

7. Once you are done using the pool of worker processes, the `Pool` instance should be closed to release resources. This is achieved by using the `with` statement, as shown in step 3. The `Pool` object automatically cleans up the worker processes when the `with` block is exited.

Here's a complete example demonstrating the usage of `pool.map`:

```
from multiprocessing import Pool

# Function to compute the square of a number
def compute_square(x):
    return x ** 2

# Create an instance of Pool with 4 worker processes
num_workers = 4
with Pool(processes=num_workers) as pool:
    # Prepare the input data as a list of numbers
    input_data = [1, 2, 3, 4, 5]

    # Apply the compute_square function to each element of the
        input data in parallel
    results = pool.map(compute_square, input_data)

# Process the results
for input_value, result in zip(input_data, results):
    print(f"The square of {input_value} is {result}")
```

In this example, the `compute_square` function calculates the square of a given number. The `Pool` object is created with 4 worker processes. The input data is a list of numbers. The `pool.map` method applies the `compute_square` function to each element of the input data in parallel, and the computed results are stored in the `results` list. Finally, the results are printed to the console.

2 Todo

Depend on your efficiency in python, you could pick either main or advanced task to implement SyncSGD.

- **Main Task - Easy:** Download the python template and fill in the missing code.
- **Main Task - Hard:** Follow the guide on previous section, use the code from previous exercise and implement your own SyncSGD framework.
- **Optional Task:** Modify your code to include momentum SGD, Adam, and Newton's Method.

Please update your code from last week's exercise to include support for Synchronous SGD, based on the information provided in the previous tutorial. Ensure that your algorithm strictly adheres to the outline provided in the lecture on page 53. Finally, execute your program using the following configuration/setup:

```
np.random.seed(0)
X = np.random.rand(1000, 2) # 100 random 2-dimensional points
y = 7 * X[:, 0] + 1 * X[:, 1] + 1*np.random.randn(1000) # Linear
    relationship with some random noise

# Set hyperparameters
batch_size = ?
learning_rate = ?
num_iterations = ?
num_workers = ? # Number of distributed workers

# Run synchronized distributed SGD
theta, theta_history = synchronized_distributed_sgd(X, y,
    batch_size, learning_rate, num_iterations, num_workers)

# Plot the loss contour and theta values
plot_loss_contour(X, y, theta_history)
```

Report your code modification and the parameters that have the best training result:

- Batch size
- Learning Rate
- Number of workers
- Training Time Cost
- Loss Contour