III. Methodology

3.1 Dataset Generation:

3.1.1 CAMB Configuration:

The dataset consists of 1,000,000 matter power spectra generated using the CAMB software (version 1.3.5). CAMB was configured to calculate the linear matter power spectrum, P(k), for a range of cosmological parameters. The following parameters were varied:

- Ω_m (matter density parameter): Uniformly sampled between 0.1 and 0.5.
- Ω_b (baryon density parameter): Uniformly sampled between 0.01 and 0.04.
- Ω_Λ (dark energy density parameter): Calculated as $\Omega_\Lambda = 1 - \Omega_m$, ensuring a flat universe (k=0).
- h (Hubble constant): Uniformly sampled between 0.5 and 0.9.
- σ_8 (amplitude of matter fluctuations): Uniformly sampled between 0.6 and 1.0.
- n_s (spectral index): Uniformly sampled between 0.9 and 1.0.

These ranges were chosen to encompass the currently favored values from cosmological observations and to provide sufficient variation for training the emulator. CAMB was configured to output the matter power spectrum at a range of wavenumbers, k, from 10^{-3} to 10^{1} h Mpc^{-1}. The number of k-points was set to 200, logarithmically spaced.

3.1.2 Sampling Strategy:

Latin Hypercube Sampling (LHS) was used to sample the six-dimensional parameter space. LHS is a stratified sampling technique that ensures a more uniform coverage of the parameter space compared to simple random sampling. This helps to avoid clustering of sample points and ensures that the emulator is trained on a representative set of cosmological models. The `scipy.stats.qmc.LatinHypercube` function was used for generating the LHS.

3.1.3 Data Preprocessing:

The following preprocessing steps were applied to the CAMB output:

- **Normalization:** The matter power spectra were normalized by dividing P(k) by the value of P(k) at the smallest k-value ($k_{min} = 10^{-3}$ h Mpc^{-1}) for each spectrum. This removes the overall amplitude dependence and focuses on the shape of the power spectrum.

- **Logarithmic Scaling:** Both the wavenumber, k, and the normalized power spectrum, $P(k)/P(k_{min})$, were transformed to logarithmic scale using base 10 logarithms. This is because both k and P(k) span several orders of magnitude, and logarithmic scaling helps to prevent numerical issues during training.
- **Data Splitting:** The dataset was split into three subsets:
  - Training set: 80% (800,000 spectra)
  - Validation set: 10% (100,000 spectra)
  - Testing set: 10% (100,000 spectra)

  The training set is used to train the network, the validation set is used for hyperparameter tuning, and the testing set is used for final evaluation of the emulator's performance. The splitting was done randomly.

3.1.4 Data Storage:

The generated data, including the cosmological parameters and the corresponding preprocessed matter power spectra, were stored in HDF5 files for efficient storage and retrieval.

3.2 Neural Network Architecture:

3.2.1 FCNN Design:

A Fully Connected Neural Network (FCNN) architecture was chosen for the emulator. The network consists of:

- **Input Layer:** 6 nodes, corresponding to the six cosmological parameters (Ω_m, Ω_b, h, σ_8, n_s).
- **Hidden Layers:** 4 hidden layers, each with 1024 nodes. This architecture was chosen based on preliminary experiments and previous work on similar emulation tasks. A relatively deep network with a large number of nodes per layer was found to be necessary to capture the complex non-linear relationship between the cosmological parameters and the matter power spectrum.
- **Output Layer:** 200 nodes, corresponding to the 200 k-values at which the normalized and log-transformed P(k) is evaluated.
- **Activation Functions:** The Rectified Linear Unit (ReLU) activation function was used for all hidden layers: ReLU(x) = max(0, x). ReLU is a common choice for deep neural networks due to its computational efficiency and ability to mitigate the vanishing gradient problem. No activation function was used for the output layer, as we are performing a regression task.

[Insert a diagram of the network architecture here. The diagram should visually represent the layers, nodes, and connections, clearly labeling the input and output layers and indicating the activation functions.]

3.3 Hyperparameter Tuning:

3.3.1 Optuna:

The Optuna framework (version 2.10.0) was used for hyperparameter optimization. Optuna is a Python-based optimization library that provides efficient algorithms for finding optimal hyperparameter values.

3.3.2 Search Space:

The following hyperparameters were tuned:

- **Learning Rate:** Sampled from a logarithmic distribution between 10^{-5} and 10^{-2}.
- **Batch Size:** Sampled from a uniform distribution between 32 and 256.
- **Optimizer:** Two optimizers were considered: Adam and SGD with momentum. For SGD, the momentum parameter was fixed at 0.9.

3.3.3 Optimization Process:

Optuna was configured to use the Tree-structured Parzen Estimator (TPE) algorithm, a Bayesian optimization method. A total of 100 trials were performed. For each trial, a set of hyperparameters was sampled from the search space, the FCNN was trained on the training set for a maximum of 100 epochs, and the Mean Squared Error (MSE) on the validation set was used as the objective function to be minimized. Optuna's pruning feature was enabled to early-stop unpromising trials, significantly reducing the computational cost.

IV. Implementation Details

4.1 Jax Framework:

4.1.1 Introduction to Jax:

Jax is a Python library developed by Google for high-performance numerical computation and machine learning research. It provides features such as automatic differentiation, just-in-time (JIT) compilation, and support for both CPU and GPU acceleration. Jax's ability to automatically differentiate through complex numerical computations makes it particularly well-suited for implementing and training neural networks.

4.1.2 Data Loading and Preprocessing in Jax:

The dataset, stored in HDF5 files, was loaded using the `h5py` library. The data preprocessing steps (normalization, logarithmic scaling, and splitting) were implemented using Jax's NumPy-compatible API (`jax.numpy`).

```
import jax.numpy as jnp
import h5py

# Load data from HDF5 file
with h5py.File('data.h5', 'r') as f:
    cosmo_params = jnp.array(f['cosmo_params'])
    power_spectra = jnp.array(f['power_spectra'])
    k_values = jnp.array(f['k_values'])

# Normalization
p_min = power_spectra[:, 0]  # P(k) at smallest k
power_spectra_normalized = power_spectra / p_min[:, None]

# Logarithmic scaling
k_values_log = jnp.log10(k_values)
power_spectra_log = jnp.log10(power_spectra_normalized)

# Data splitting (example for training set)
train_indices = ... # Indices for training set
cosmo_params_train = cosmo_params[train_indices]
power_spectra_train = power_spectra_log[train_indices]
```

4.1.3 FCNN Implementation in Jax:

The FCNN architecture was implemented using Jax's `stax` library, which provides building blocks for creating neural networks.

```
from jax import random
from jax.experimental import stax
from jax.experimental.stax import Dense, Relu, LogSoftmax

# Define the network architecture
init_fun, apply_fun = stax.serial(
    Dense(1024), Relu,
    Dense(1024), Relu,
    Dense(1024), Relu,
    Dense(1024), Relu,
    Dense(200)  # Output layer
)

# Initialize network parameters
rng = random.PRNGKey(0)
input_shape = (-1, 6)  # Batch size x number of cosmological parameters
output_shape, params = init_fun(rng, input_shape)
```

4.1.4 Loss Function and Optimizer in Jax:

The Mean Squared Error (MSE) loss function and the chosen optimizer (Adam or SGD) were implemented using Jax.

```
from jax.experimental import optimizers

def mse_loss(params, batch):
    inputs, targets = batch
    predictions = apply_fun(params, inputs)
    return jnp.mean((predictions - targets)**2)

# Example: Adam optimizer
learning_rate = ... # Optimized learning rate
opt_init, opt_update, get_params = optimizers.adam(learning_rate)
opt_state = opt_init(params)
```

4.1.5 Training Loop in Jax:

The training loop iterates over the training data in batches, calculates the loss, computes the gradients using Jax's `grad` function, and updates the network parameters using the optimizer.

```
from jax import jit, grad

@jit
def update(i, opt_state, batch):
    params = get_params(opt_state)
    loss = mse_loss(params, batch)
    grads = grad(mse_loss)(params, batch)
    return opt_update(i, grads, opt_state), loss

num_epochs = 100
batch_size = ... # Optimized batch size

for epoch in range(num_epochs):
    for batch_idx in range(num_batches):
        batch = ... # Get batch of data
        opt_state, loss = update(epoch * num_batches + batch_idx, opt_state, batch)
        print(f"Epoch {epoch}, Batch {batch_idx}, Loss: {loss}")

# Get final trained parameters
trained_params = get_params(opt_state)
```

4.2 Computational Resources:

4.2.1 Cambridge HPC:

The computations were performed on the Cambridge Service for Data Driven Discovery (CSD3) HPC system. Both CPU and GPU resources were utilized. Specifically, NVIDIA Tesla V100 GPUs were used for training the neural networks, and Intel Xeon Skylake CPUs were used for data generation and preprocessing.

4.2.2 Software Environment:

The following software environment was used:

- Python 3.8
- Jax 0.2.25
- CAMB 1.3.5
- Optuna 2.10.0
- h5py 3.6.0
- NumPy 1.21.4
- SciPy 1.7.3

4.3 Data Interpolation:

4.3.1 Need for Interpolation:

CAMB outputs the matter power spectrum at a set of k-values that can vary slightly depending on the input cosmological parameters. To train the neural network, we need the power spectra to be evaluated at a fixed, consistent set of k-values for all parameter combinations. Therefore, interpolation is necessary.

4.3.2 Interpolation Method:

Linear interpolation was chosen for its simplicity and computational efficiency. Given two points (k_i, $P(k_i)$) and (k_{i+1}, $P(k_{i+1})$), the interpolated value P(k) at a point k between k_i and k_{i+1} is calculated as:

```
P(k) = P(k<sub>i</sub>) + (k - k<sub>i</sub>) * (P(k<sub>i+1</sub>) - P(k<sub>i</sub>)
```

Since we are working with log-transformed values, the interpolation is performed on log(k) and log(P(k)).

4.3.3 Implementation in Jax:

The linear interpolation was implemented using Jax's `jnp.interp` function.

```
# k_values_fixed: Fixed set of k-values for the emulator
# k_values_camb: k-values from CAMB output for a specific parameter combination
# power_spectrum_camb: P(k) from CAMB output for the same parameter combination

power_spectrum_interpolated = jnp.interp(jnp.log10(k_values_fixed), jnp.log10(k_values
```

This interpolation step is performed within the data preprocessing pipeline before training the network.