# JAX

The purpose of this exercise is to demonstrate the core functionalities of the JAX library in Python for machine learning, including basic operations, automatic differentiation, JIT compilation, automatic vectorization, and automatic parallelization, and a simple example of building a neural network with JAX.

# Introduction to JAX

JAX is a Python library designed for high-performance numerical computing, particularly suited for machine learning research. It combines a familiar NumPy-like API with powerful features for transforming numerical functions.

## Key Features of JAX for Machine Learning

- **NumPy Compatibility:** JAX provides a NumPy-like API, making it easy for users familiar with NumPy to transition and leverage its advanced features. This allows for writing concise and readable code.
- **Automatic Differentiation (Gradients):** JAX can automatically compute derivatives of Python and NumPy functions. This is crucial for training machine learning models using gradient-based optimization algorithms like stochastic gradient descent (SGD). The `jax.grad` function enables efficient computation of gradients.
- **JIT Compilation (Just-In-Time Compilation):** JAX can Just-In-Time compile Python functions using XLA (Accelerated Linear Algebra), a compiler for linear algebra that can target various hardware accelerators like GPUs and TPUs. This compilation significantly speeds up the execution of numerical computations by optimizing the code for the underlying hardware. The `jax.jit` decorator is used for this purpose.
- **Automatic Vectorization (vmap):** JAX provides automatic vectorization using the `jax.vmap` function. This allows you to automatically batch computations across a new axis, eliminating the need for explicit batching loops and improving performance, especially on accelerators. This is particularly useful for processing batches of data in machine learning.
- **Automatic Parallelization (pmap):** For multi-core or multi-device setups, JAX offers automatic parallelization with the `jax.pmap` function. This allows you to easily parallelize computations across multiple devices (e.g., multiple GPUs or TPU cores), enabling faster training of large models on distributed hardware.

## Jax basics

### Subtask:

Demonstrate basic JAX operations, including working with JAX arrays and automatic differentiation using `jax.grad`.

In [ ]:
```python
import jax
import jax.numpy as jnp

# Create a JAX array from a Python list
jax_array_list = jnp.array([1.0, 2.0, 3.0])
print("JAX array from list:", jax_array_list)

# Create a JAX array from a NumPy array
import numpy as np
numpy_array = np.array([4.0, 5.0, 6.0])
jax_array_numpy = jnp.array(numpy_array)
print("JAX array from NumPy array:", jax_array_numpy)

# Perform a basic arithmetic operation
result_addition = jax_array_list + jax_array_numpy
print("Result of addition:", result_addition)

# Define a Python function for differentiation
def square_and_sum(x):
  return jnp.sum(x**2)

# Use jax.grad to compute the gradient
input_for_grad = jnp.array([2.0, 4.0, 6.0])
gradient_function = jax.grad(square_and_sum)
computed_gradient = gradient_function(input_for_grad)

# Print the input to the gradient function and the computed gradient
print("Input to gradient function:", input_for_grad)
print("Computed gradient:", computed_gradient)

# Our function in square_and_sum() is x^2, so its derivative
# from jax.grad() will be 2x.
```

```
JAX array from list: [1. 2. 3.]
JAX array from NumPy array: [4. 5. 6.]
Result of addition: [5. 7. 9.]
Input to gradient function: [2. 4. 6.]
Computed gradient: [ 4.  8. 12.]
```

# JIT Compilation with `jax.jit`

Just-In-Time (JIT) compilation is a technique used to improve the performance of programs by compiling code during runtime rather than before execution. In the context of JAX, JIT compilation is performed by XLA (Accelerated Linear Algebra), Google's domain-specific compiler for linear algebra that can target various hardware accelerators like GPUs and TPUs.

The `jax.jit` function (often used as a decorator) is the primary tool in JAX for applying JIT compilation to Python functions. When a function decorated with `jax.jit` is called for the first time with a particular set of argument shapes and dtypes, JAX traces the function's execution, builds a computation graph, and compiles this graph using XLA. Subsequent calls to the same function with arguments of the same shapes and dtypes will reuse the compiled code, leading to significantly faster execution, especially for computationally intensive operations.

Key benefits of using `jax.jit` include:

- **Performance Improvement:** By compiling and optimizing the computation graph, JAX can leverage the underlying hardware more efficiently, leading to substantial speedups.
- **Reduced Python Overhead:** JIT compilation moves the execution from Python's interpreter to optimized compiled code, reducing the overhead associated with Python loops and function calls.
- **Hardware Acceleration:** XLA can compile code for various accelerators, allowing JAX to efficiently run computations on GPUs and TPUs.

It's important to note that `jax.jit` works best on functions that primarily operate on JAX arrays and have a consistent structure (control flow that depends on the *values* of inputs can be tricky).

**Reasoning**: Define a simple Python function for numerical computation, measure its execution time without JIT using `%timeit`, apply `jax.jit`, and measure the execution time again with `%timeit` for comparison.

In [ ]:
```python
import jax
import jax.numpy as jnp
import numpy as np

# Define a simple Python function
def simple_computation(x):
  return jnp.dot(x, x)

# Create a large JAX array
large_array = jnp.ones(1000000)

# Measure execution time without JIT compilation
print("Measuring execution time without JIT:")
%timeit simple_computation(large_array)

# Apply jax.jit to the function
@jax.jit
def simple_computation_jit(x):
  return jnp.dot(x, x)

# Measure execution time with JIT compilation
# The first call will include compilation time, subsequent calls will use the compiled code.
# We run it once outside timeit to trigger compilation.
_ = simple_computation_jit(large_array)

print("\nMeasuring execution time with JIT compilation:")
%timeit simple_computation_jit(large_array)
```

```
Measuring execution time without JIT:
127 µs ± 52.1 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Measuring execution time with JIT compilation:
109 µs ± 29.7 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

# Automatic Vectorization with `jax.vmap`

Automatic vectorization is a technique that allows a function designed to operate on a single data point to be efficiently applied to a batch of data. In the context of JAX, this is achieved using the `jax.vmap` function.

Traditionally, to apply a function element-wise to a batch, you might use a loop or explicit batching operations. However, loops in Python can be slow, and explicit batching can make the code more complex. `jax.vmap` automates this process by taking a function and returning a new function that maps the original function across an arbitrary axis of the input arrays.

Here's how `jax.vmap` works:

1. You define a function that operates on a single example (e.g., a single vector or scalar).
2. You use `jax.vmap` to create a "vectorized" version of this function.
3. When you call the vectorized function with a batch of data, `vmap` automatically handles the looping and batching internally, applying the original function to each element (or slice) along the specified axis of the input batch.

Benefits of using `jax.vmap`:

- **Performance:** `vmap` is designed to be highly efficient, especially when combined with `jax.jit`, as it allows JAX to optimize the batched computation.
- **Simplified Code:** It eliminates the need for explicit loops or manual batching logic, making your code cleaner and easier to read.
- **Flexibility:** `vmap` can map over arbitrary axes, providing flexibility in how you process your data batches.

This is particularly useful in machine learning for applying operations to batches of training examples, where the same function needs to be applied independently to each example in the batch.

In [ ]:
```python
import jax.numpy as jnp
import jax

# 2. Define a simple JAX function that operates on a single input
def apply_transformation(x):
  """Applies a simple transformation: squares the input and adds 5."""
  return x**2 + 5

# 3. Create a batch of input data using JAX arrays
batch_input = jnp.array([1.0, 2.0, 3.0, 4.0, 5.0])
print("Original batch input:", batch_input)

# 4. Apply the function defined in step 2 to the batch of data using jax.vmap
# vmap will automatically map the apply_transformation function across the first axis (axis 0) of the batch_input.
vectorized_transformation = jax.vmap(apply_transformation)
batch_output = vectorized_transformation(batch_input)

# 5. Print the input batch and the result of the vmap-ped function
print("Result of vmap-ped transformation:", batch_output)
```

```
Original batch input: [1. 2. 3. 4. 5.]
Result of vmap-ped transformation: [ 6.  9. 14. 21. 30.]
```

# Automatic Parallelization with `jax.pmap`

Automatic parallelization is the process of automatically distributing computations across multiple processing units (like CPU cores, GPUs, or TPU cores) to speed up execution. For large-scale machine learning tasks, such as training deep neural networks on massive datasets, leveraging multiple devices is essential to achieve reasonable training times.

JAX provides `jax.pmap` (parallel map) to enable automatic parallelization across multiple devices. Similar to `jax.vmap` which maps a function across a batch dimension, `jax.pmap` maps a function across the devices available to JAX. It is designed to work with the **SPMD (Single Program, Multiple Data)** programming model. In the SPMD model, the same program (or function) is executed on multiple devices, but each device operates on a different slice or portion of the data.

Here's how `jax.pmap` works:

1. You define a JAX function that operates on a single device's slice of data.
2. You use `jax.pmap` to create a parallelized version of this function.
3. When you call the parallelized function, JAX automatically distributes the input data across the available devices and executes the function on each device in parallel.
4. The results from each device are then typically collected or synchronized, depending on the specific computation.

`jax.pmap` is particularly useful for:

- **Data Parallelism:** Distributing batches of data across multiple devices, where each device processes a different sub-batch and computes gradients in parallel.
- **Model Parallelism (limited):** In some cases, parts of a model can be placed on different devices, although `pmap` is primarily designed for data parallelism.

It's important to structure your data and function such that the computation can be effectively split and run independently on each device. `jax.pmap` handles the complexities of distributing the computation and managing communication between devices.

**Task**: Demonstrate `jax.pmap` by defining a simple function, parallelizing it with `pmap`, creating distributed input data, and executing the parallelized function.

In [ ]:
```python
import jax
import jax.numpy as jnp

# 1. Define a simple JAX function that can be executed in parallel
# This function operates on a single slice of data for a device
def device_computation(x):
  """A simple function to be executed on each device: squares the input."""
  print(f"Executing on device: {jax.devices()[jax.process_index()]}")
  return x**2

# 2. Use jax.pmap to create a parallelized version of the function
# This maps the function across the available devices
parallelized_computation = jax.pmap(device_computation)

# Get the number of available devices
num_devices = jax.local_device_count()
print(f"\nNumber of available devices: {num_devices}")

# 3. Create input data that can be distributed across devices.
# We'll create a single array and split it across the leading dimension for pmap.
# The size of the leading dimension must be equal to the number of devices.
if num_devices > 0:
  # Create dummy data with a leading dimension equal to the number of devices
  input_data = jnp.arange(num_devices * 5).reshape(num_devices, 5)
  print("\nInput data shape for pmap:", input_data.shape)
  print("Input data distributed across devices:")
  # Print data for each device
  for i in range(num_devices):
      print(f"  Device {i}: {input_data[i]}")

  # 4. Execute the parallelized function with the input data.
  # JAX will automatically distribute the first dimension of input_data across devices.
  output_data = parallelized_computation(input_data)

  # 5. Print the output from the parallelized function
  print("\nOutput data shape from pmap:", output_data.shape)
  print("Output data from parallelized computation (collected from devices):")
  print(output_data)
else:
  print("\nNo JAX devices found. Cannot demonstrate jax.pmap.")
```

```
Number of available devices: 1

Input data shape for pmap: (1, 5)
Input data distributed across devices:
  Device 0: [0 1 2 3 4]
Executing on device: cuda:0

Output data shape from pmap: (1, 5)
Output data from parallelized computation (collected from devices):
[[ 0  1  4  9 16]]
```

## Jax and neural networks

**Task**: Building a simple neural network with JAX and using JAX transformations like `jax.grad`, define the necessary functions for a neural network (ReLU, prediction, loss), generate fake data, initialize parameters, compute loss and gradients, etc.

In [ ]:
```python
import jax
import jax.numpy as jnp
import jax.random as jrandom

# 2. Define a function `relu` for the ReLU activation function
def relu(x):
  return jnp.maximum(0, x)

# 3. Define a function `predict` for a simple two-layer neural network
def predict(params, x):
  # params is a list of (weight, bias) tuples for each layer
  w1, b1 = params[0]
  w2, b2 = params[1]

  # First layer
  hidden = relu(jnp.dot(x, w1) + b1)

  # Output layer
  output = jnp.dot(hidden, w2) + b2
  return output

# 4. Define a loss function, e.g., mean squared error (mse_loss)
def mse_loss(params, x, y):
  predictions = predict(params, x)
  return jnp.mean((predictions - y)**2)

# 5. Define a function `generate_fake_data`
def generate_fake_data(key, num_samples, input_size, output_size):
  # Generate random input features
  x = jrandom.normal(key, (num_samples, input_size))
  # Generate corresponding labels based on a simple linear relationship with noise
  true_weights = jrandom.normal(key, (input_size, output_size))
  true_bias = jrandom.normal(key, (output_size,))
  noise = jrandom.normal(key, (num_samples, output_size)) * 0.1
  y = jnp.dot(x, true_weights) + true_bias + noise
  return x, y

# 6. Initialize the network parameters (weights and biases)
key = jrandom.PRNGKey(0)
key, subkey1, subkey2, subkey3, subkey4, data_key = jrandom.split(key, 6)

input_size = 10
hidden_size = 20
output_size = 1

# Initialize weights and biases for two layers
w1 = jrandom.normal(subkey1, (input_size, hidden_size)) * 0.01 # Initialize with small values
b1 = jrandom.normal(subkey2, (hidden_size,)) * 0.01
w2 = jrandom.normal(subkey3, (hidden_size, output_size)) * 0.01
b2 = jrandom.normal(subkey4, (output_size,)) * 0.01

initial_params = [(w1, b1), (w2, b2)]

# 7. Generate fake data
num_samples = 100
x_data, y_data = generate_fake_data(data_key, num_samples, input_size, output_size)

# 8. Compute the gradient of the `mse_loss` function with respect to the network parameters
# jax.grad takes the function to differentiate and the argument index(es) to differentiate with respect to.
# Here, we want to differentiate with respect to the 'params' argument, which is the first argument (index 0).
grad_loss = jax.grad(mse_loss, argnums=0)

# 9. Demonstrate calculating the loss and the gradients for the initial parameters and fake data
initial_loss = mse_loss(initial_params, x_data, y_data)
initial_grads = grad_loss(initial_params, x_data, y_data)

print(f"Initial Loss: {initial_loss}")
print("\nInitial Gradients (for w1, b1, w2, b2):")
# Print shapes of gradients to show they match parameter shapes
print(f"  grad_w1 shape: {initial_grads[0][0].shape}")
print(f"  grad_b1 shape: {initial_grads[0][1].shape}")
print(f"  grad_w2 shape: {initial_grads[1][0].shape}")
print(f"  grad_b2 shape: {initial_grads[1][1].shape}")

# 10. (Optional) Wrap the loss and gradient computation with jax.jit
# This will compile the functions for performance
jit_mse_loss = jax.jit(mse_loss)
jit_grad_loss = jax.jit(grad_loss)

# Demonstrate calling the JIT-compiled functions
# The first call will incur compilation overhead
jit_initial_loss = jit_mse_loss(initial_params, x_data, y_data)
jit_initial_grads = jit_grad_loss(initial_params, x_data, y_data)

print("\n--- JIT-compiled results ---")
print(f"JIT Initial Loss: {jit_initial_loss}")
print("\nJIT Initial Gradients (for w1, b1, w2, b2):")
print(f"  jit_grad_w1 shape: {jit_initial_grads[0][0].shape}")
print(f"  jit_grad_b1 shape: {jit_initial_grads[0][1].shape}")
print(f"  jit_grad_w2 shape: {jit_initial_grads[1][0].shape}")
print(f"  jit_grad_b2 shape: {jit_initial_grads[1][1].shape}")
```

```
Initial Loss: 10.784708023071289

Initial Gradients (for w1, b1, w2, b2):
  grad_w1 shape: (10, 20)
  grad_b1 shape: (20,)
  grad_w2 shape: (20, 1)
  grad_b2 shape: (1,)

--- JIT-compiled results ---
JIT Initial Loss: 10.784708023071289

JIT Initial Gradients (for w1, b1, w2, b2):
  jit_grad_w1 shape: (10, 20)
  jit_grad_b1 shape: (20,)
  jit_grad_w2 shape: (20, 1)
  jit_grad_b2 shape: (1,)
```

## Summary:

- JAX provides a NumPy-like API for numerical computing, making it accessible to users familiar with NumPy.
- JAX supports automatic differentiation using `jax.grad`, allowing for efficient computation of gradients of Python and NumPy functions.
- JIT compilation with `jax.jit` significantly speeds up the execution of numerical computations by compiling and optimizing code using XLA.
- Automatic vectorization with `jax.vmap` enables the efficient application of a function to batches of data without explicit loops, simplifying code and improving performance.
- Automatic parallelization with `jax.pmap` facilitates the distribution of computations across multiple devices using the SPMD model, which is crucial for large-scale machine learning.
- JAX can be used to build and train neural networks by defining model architecture, loss functions, and leveraging `jax.grad` for gradient computation and `jax.jit` for performance optimization.

In [ ]: