Title: Generic Machine Learning Library in C++ (GMLL)

Objective:
Develop a generic machine learning library in C++ that is inspired by Keras. The library should be able to create Deep Neural Networks (DNN), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN). The library must be callable in Python.
Requirements:
Use C++ as the primary programming language.
Implement template classes and functions to ensure generic programming.
Provide an interface to call the library from Python.
Do not use any third-party libraries like Boost or Eigen.
Code must be clean, well-organized, and well-documented.
Classes and Functions Signatures:
Tensor class:
```
template<typename T>
class Tensor {
public:
   Tensor(std::vector<size_t> shape);
   T& operator()(size_t i, size_t j, size_t k);
   const T& operator()(size_t i, size_t j, size_t k) const;
   // Additional member functions and operators as needed
};
```
Layer interface:
```
template<typename T>
class Layer {
public:
   virtual void forward(const Tensor<T>& input, Tensor<T>& output) = 0;
   virtual void backward(const Tensor<T>& input, const Tensor<T>& output_gradient, Tensor<T>&
input_gradient) = 0;
   virtual void update_weights(float learning_rate) = 0;
};
```
Dense layer:
```
template<typename T>
class Dense : public Layer<T> {
public:
   Dense(size_t input_size, size_t output_size);
   void forward(const Tensor<T>& input, Tensor<T>& output) override;
   void backward(const Tensor<T>& input, const Tensor<T>& output_gradient, Tensor<T>&
input_gradient) override;
   void update_weights(float learning_rate) override;
};
```
Convolution layer:
```
template<typename T>
class Convolution : public Layer<T> {
public:
   Convolution(size_t input_channels, size_t output_channels, size_t kernel_size);
   void forward(const Tensor<T>& input, Tensor<T>& output) override; void backward(const
Tensor<T>& input, const Tensor<T>& output_gradient, Tensor<T>& input_gradient) override;
```

```
    void update_weights(float learning_rate) override;
};
```

Recurrent layer:

```
template<typename T>
class Recurrent : public Layer<T> {
public:
    Recurrent(size_t input_size, size_t output_size);
    void forward(const Tensor<T>& input, Tensor<T>& output) override;
    void backward(const Tensor<T>& input, const Tensor<T>& output_gradient, Tensor<T>&
input_gradient) override;
    void update_weights(float learning_rate) override;
};
```

Model class:

```
template<typename T>
class Model {
public:
    void add_layer(Layer<T>* layer);
    void compile(const std::string& loss);
    void fit(const Tensor<T>& input, const Tensor<T>& target, size_t batch_size, size_t epochs, float
learning_rate);
    void predict(const Tensor<T>& input, Tensor<T>& output);
};
```

Sample code (DNN):

Here's an example of how the library will be used to create a simple Deep Neural Network:

```
#include "gml.h"

int main() {
    // Create a model
    Model<float> model;

    // Add layers to the model
  model.add_layer(new Dense<float>(784, 128));
model.add_layer(new Dense<float>(128, 64));
model.add_layer(new Dense<float>(64, 10));

// Compile the model with a loss function
model.compile("categorical_crossentropy");

// Train the model
Tensor<float> input_data(/*input data shape*/);
Tensor<float> target_data(/*target data shape*/);
model.fit(input_data, target_data, 32, 10, 0.001);

// Make predictions
Tensor<float> new_input_data(/*input data shape*/);
Tensor<float> output_data(/*output data shape*/);
model.predict(new_input_data, output_data);

return 0;
}
```

Python Interface:

Use `pybind11` to create a Python interface for the GMLL library. Here's an example of how to create the binding for the `Model` class:

~~~
```//
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include "gml.h"
namespace py = pybind11;
PYBIND11_MODULE(gmll, m) {
  py::class_<Model<float>>(m, "Model")
    .def(py::init<>())
    .def("add_layer", &Model<float>::add_layer)
    .def("compile", &Model<float>::compile)
    .def("fit", &Model<float>::fit)
    .def("predict", &Model<float>::predict);
}
~~~

Additional Function Signatures:

Activation functions:

```
template<typename T>
T sigmoid(T x);

template<typename T>
T relu(T x);

template<typename T>
T softmax(T x);

template<typename T>
T tanh(T x);
```

Loss functions interface:

```
template<typename T>
class LossFunction {
public:
  virtual T compute(const Tensor<T>& predicted, const Tensor<T>& target) = 0;
  virtual void compute_gradient(const Tensor<T>& predicted, const Tensor<T>& target, Tensor<T>&
gradient) = 0;
};
```

Categorical cross-entropy loss:

```
template<typename T>
class CategoricalCrossentropy : public LossFunction<T> {
public:
   T compute(const Tensor<T>& predicted, const Tensor<T>& target) override;
   void compute_gradient(const Tensor<T>& predicted, const Tensor<T>& target, Tensor<T>&
gradient) override;
};
```

Mean squared error loss:

```
template<typename T>
class MeanSquaredError : public LossFunction<T> {
public:
   T compute(const Tensor<T>& predicted, const Tensor<T>& target) override;
   void compute_gradient(const Tensor<T>& predicted, const Tensor<T>& target, Tensor<T>&
gradient) override;
};
```

Create a loss function factory:

```
template<typename T>
std::unique_ptr<LossFunction<T>> create_loss_function(const std::string& name);
```

Update the Model class compile function to use the loss function factory:

```
template<typename T>
class Model {
public:
   // ...
   void compile(const std::string& loss) {
      loss_function = create_loss_function<T>(loss);
   }
   // ...
};
```

Unit Tests:

Create unit tests for the following functionalities:

Test each layer type (Dense, Convolution, Recurrent) for forward and backward propagation.
Test activation functions for correctness (sigmoid, relu, softmax, tanh).
Test loss functions for correctness (categorical cross-entropy, mean squared error).
Test model fitting and prediction functionalities.
Test Python interface functionality.
Ensure that all tests cover various edge cases and input combinations to validate the correctness of the library.

Deliverables:

Updated source code files (header and implementation files) for all classes and functions.
Updated Python interface files using pybind11.
Updated sample code demonstrating the use of the library for creating DNN, CNN, and RNN models.
Unit tests to validate the correctness of the library.

Updated README file containing instructions on how to build, test, and use the library.
Detailed documentation of the library, including an explanation of the classes, functions, and their parameters.

Optimization Algorithms:

Implement common optimization algorithms such as Stochastic Gradient Descent (SGD), Adam, and RMSprop. Create an Optimizer interface:

```
template<typename T>
class Optimizer {
public:
   virtual void update(Tensor<T>& weights, const Tensor<T>& gradients, float learning_rate) = 0;
};
```

Stochastic Gradient Descent (SGD) optimizer:

```
template<typename T>
class SGD : public Optimizer<T> {
public:
   void update(Tensor<T>& weights, const Tensor<T>& gradients, float learning_rate) override;
};
```

Adam optimizer:

```
template<typename T>
class Adam : public Optimizer<T> {
public:
   Adam(float beta1 = 0.9, float beta2 = 0.999, float epsilon = 1e-8);
   void update(Tensor<T>& weights, const Tensor<T>& gradients, float learning_rate) override;

private:
   float beta1, beta2, epsilon;
   Tensor<T> m, v;
   size_t t;
};
```

RMSprop optimizer:

```
template<typename T>
class RMSprop : public Optimizer<T> {
public:
   RMSprop(float decay_rate = 0.9, float epsilon = 1e-8);
   void update(Tensor<T>& weights, const Tensor<T>& gradients, float learning_rate) override;

private:
   float decay_rate, epsilon;
   Tensor<T> cache;
};
```

Create an optimizer factory:

```
template<typename T>
std::unique_ptr<Optimizer<T>> create_optimizer(const std::string& name, const
std::unordered_map<std::string, float>& params = {});
```

Update the Model class to use the optimizer factory:

```
template<typename T>
class Model {
public:
   // ...
   void compile(const std::string& loss, const std::string& optimizer) {
      loss_function = create_loss_function<T>(loss);
      optimizer_instance = create_optimizer<T>(optimizer);
   }
   // ...
};
```

Unit Tests:
Extend the existing unit tests to cover the following functionalities:
Test optimizer algorithms for correctness (SGD, Adam, RMSprop).
Deliverables:
Updated source code files (header and implementation files) for all classes and functions, including optimization algorithms.
Updated Python interface files using pybind11.
Updated sample code demonstrating the use of the library for creating DNN, CNN, and RNN models.
Unit tests to validate the correctness of the library, including tests for optimization algorithms.
Updated README file containing instructions on how to build, test, and use the library.
Detailed documentation of the library, including an explanation of the classes, functions, and their parameters.
With these additional functionalities, your Generic Machine Learning Library will be more versatile and robust, allowing users to easily create various types of neural networks, apply different loss functions and optimization algorithms, and interface with Python for more seamless integration in a wider variety of projects.
Guidelines on programming practices and submission requirements.
A comprehensive set of unit tests for validating the library's correctness.

Last date for submission is the ISA5 last day (Elective 4)+ 2 days.