PLearn Programmer's Guide

A programmer's view of the Plearn C++ Machine-Learning Library and tools

August 3, 2022

Contents

Chapter 1

Overview of PLearn

1.1 Introduction

Machine Learning algorithms are usually described in scientific papers in a standard mathematical formulation, often framed as an optimization of a given cost function. PLearn is a C++ library that uses the object-oriented and operator overloading capabilities of the C++ language to allow, among other things, to express cost functions and their optimization as a standard C++ program, in a declarative manner that is as close as possible to their mathematical formalization.

Most neural-network and general machine-learning simulation environments define their own scripting language. While it is very tempting for every computer-scientist to craft his own language, creating a complete, clear, efficient and bug-free language is a horrendous task, so this is how things usually go: one starts bulding a simple scripting syntax (typically lisp-like because it's easy to parse) to specify simple experiments. Quickly it appears too limited, and it may grow to include loops, functions, data structures, etc. Eventually it ends up including some sort of support for object-oriented programming, and finally for efficiency you want it to be compiled rather than interpreted! In the end, you end up with a huge mess of a system that was not designed to grow that much from the beginning, and which is often impossible to comprehend and maintain for anybody but its author. The end-result might sometimes be impressive, but at the cost of a lot of efforts diverted from your actual research. While C++ is far from being the perfect language, it is very powerful, can be both very expressive and generate highly efficient code, and most of all it has the immense advantage of being developed and well-supported by worldwide teams of dedicated and competent people...

When providing the correct type abstractions, C++ can be an expressive-enough language to directly serve as a highly customizable, extensible and efficient "scripting language" for designing and running even the most demanding experiments in machine-learning research and development. So this is what this library is all about: providing the right type abstractions. What originally got me started on this project was the desire to be able to optimize a complex cost function by just expressing it in a declarative way as close as possible to the mathematical formulation. This lead to the original implementation of the Var class. Since then PLearn has grown to include many other useful types and abstractions.

While PLearn has been successfully used by several people for over a year, it is still very much work in progress. As its primary use is for our own research, we did not want to carve it in stone: thus future versions may look quite different from this one, as we are still reworking the class hierarchy. But it is nevertheless already very usable, so feel free to play around and experiment with it!

Probably the biggest problem, like with many projects of this kind, is the cruel lack of documentation. This manual will attempt to give you a high-level understanding of the basic concepts, but to work out the details, you'll have to look at the actual code. Also, to fully use the potential of this library, you are expected to be somewhat comfortable with the C++ language.

Have fun!

Pascal

1.2 Additional tools for developers

In addition, if you wish to develop new learning algorithms, or otherwise contribute to the librery, the following tools will be useful:

- **ssh** for write access to the SourceForge CVS repository.
- **gdb** for basic debugging (or a *better* debugger if you have one!)
- **valgrind** a wonderful free tool for memory-bug hunting.
- **python** for running python scripts, as well as the PLearn build system
- **perl** for running perl scripts
- **LaTeX, pdflatex, dvips, latex2html, doxygen** to re-generate the documentation.

Chapter 2

Basics

2.1 PLearn for Matrix-Vectors Operations

PLearn has its own vector and matrix data structures. The files `PLearn/plearn/math/TVec_{decl,math}.h` contain the declaration and implementation of the vector class template`TVec`, and the matrix class template `TMat` can be found in files `PLearn/plearn/math/TMat_{decl,math}.h`.

2.1.1 Creation and Basic Manipulations

The PLearn vector and matrix data structures are easy to instantiate and support many useful basic operations, such as subvector and submatrix access.

Here is a concrete example of how to use these data structures. The data type `Vec` and `Mat` refer to the `TVec<real>` and `TMat<real>` classes, where `real` is a macro corresponding either to `double` or `float`, depending on the compilation options used.

```
#include <plearn/math/TMat_maths.h>
using namespace PLearn;

int main(int argc, char** argv)
{
    // Example use of a 'real' variable
    // a compilation option makes it either a double or a float
    // Please don't use double nor float

    real a=15;
    cout<<"a="<<a<<endl;
    // Output:
    // a=15

    // Vector creation
    Vec b(3);
    b[0] = 2;
```

```
b[1] = 42;
b[2] = 21;
cout<<"b="<<b<<endl;
cout<<"b.length()="<<b.length()<<endl;
// Output:
// b=2           42          21
// b.length()=3

// Vector manipulations:

// Subvector access of the last two elements (not a copy!!!)
Vec b3 = b.subVec(1,2);
cout<<"b3="<<b3<<endl;
// Output:
// b3=42          21

// Concatenation
Vec b4 = concat(b,b);
cout<<"b4="<<b4<<endl;
// Output:
// b4=2           42          21          2           42          21

// Note: "=" operator does not copy!!!
Vec b5 = b4;
b5[1] = 100000;
cout<<"b4="<<b4<<endl;
cout<<"b5="<<b5<<endl;
// Output:
// b4=2           100000      21          2           42          21
// b5=2           100000      21          2           42          21


// Copy
b5 = b4.copy();
b5[1]=1;
cout<<"b4="<<b4<<endl;
cout<<"b5="<<b5<<endl;
// Output:
// b4=2           100000      21          2           42          21
// b5=2           1      21          2           42          21


// Fill in one element
Vec b6(b4.length());
b6.fill(3);
cout<<"b6="<<b6<<endl;
// Output:
```

```
// b6=3           3           3           3           3           3

// Fill in elements of another vector
b6 << b4;
cout<<"b6="<<b6<<endl;
// Output:
// b6=2           100000      21          2           42          21

// Clear
b6.clear();
cout<<"b6="<<b6<<endl;
// Output:
// b6=0           0           0           0           0           0

// Resize
b4.resize(7);
b4[6] = 6;
cout<<"b4="<<b4<<endl;
// Output:
// b4=2           100000      21          2           42          21          6

b4.resize(4);
cout<<"b4="<<b4<<endl;
// Output:
// b4=2           100000      21          2


// Matrix creation :
Mat c(3,2);
c(1,1)=1.1;
c(1,0)=4;
c(2,0)=5;
c(0,1)=-73.2;
c(0,0)=78;
c(2,1)=5.32e-2;
cout<<"c=\n"<<c<<endl;
cout<<"c.length()="<<c.length()<<endl;
cout<<"c.width()="<<c.width()<<endl;
// Output:
// c=
// 78          -73.2
// 4           1.1
// 5           0.0532
//
// c.length()=3
// c.width()=2
```

```
// Matrix manipulation:

// Submatrix access (not a copy!!!)...

// ... of the last two rows and first column
Mat c3 = c.subMat(1,0,2,1);
cout<<"c3=\n"<<c3<<endl;
// Output:
// c3=
// 4
// 5

// ... of the second column
Mat c4 = c.column(1);
cout<<"c4=\n"<<c4<<endl;
// Output:
// c4=
// -73.2
// 1.1
// 0.0532

// ... of the third row
Mat c5 = c.row(2);
cout<<"c5=\n"<<c5<<endl;
// Output:
// c5=
// 5           0.0532

// .. of the third row, as a vector
Vec b7 = c(2);
cout<<"b7="<<b7<<endl;
// Output:
// b7=5           0.0532

// Note: "=" operator does not copy!!!
Mat c6 = c;
c6(1,1) = 100000;
cout<<"c=\n"<<c<<endl;
cout<<"c6=\n"<<c6<<endl;
// Output:
// c=
// 78          -73.2
// 4           100000
// 5           0.0532
//
// c6=
// 78          -73.2
```

```
// 4            100000
// 5            0.0532

// Copy
c6 = c.copy();
c6(1,1) = 1;
cout<<"c=\n"<<c<<endl;
cout<<"c6=\n"<<c6<<endl;
// Output:
// c=
// 78           -73.2
// 4            100000
// 5            0.0532
//
// c6=
// 78           -73.2
// 4            1
// 5            0.0532

// Fill in one element
Mat c7(c.length(),c.width());
c7.fill(3);
cout<<"c7=\n"<<c7<<endl;
// Output:
// c7=
// 3            3
// 3            3
// 3            3

// Fill in elements of another matrix
c7 << c;
cout<<"c7=\n"<<c7<<endl;
// Output:
// c7=
// 78           -73.2
// 4            100000
// 5            0.0532

// Fill in a row of another matrix
c7(2) << c(1);
cout<<"c7=\n"<<c7<<endl;
// Output:
// c7=
// 78           -73.2
// 4            100000
// 5            0.0532
```

```
    // Clear
    c7.clear();
    cout<<"c7=\n"<<c7<<endl;
    // Output:
    // c7=
    // 0           0
    // 0           0
    // 0           0

    // Resize
    c7.resize(4,4);
    c7.subMat(0,2,3,2)<<c.subMat(0,0,3,2);
    c7(3,0)=0.01;
    c7(3,1)=0.02;
    c7(3,2)=0.03;
    c7(3,3)=0.04;
    cout<<"c7=\n"<<c7<<endl;
    // Output:
    // c7=
    // 0           0           78          -73.2
    // 0           0           4           100000
    // 0           0           5           0.0532
    // 0.01        0.02        0.03        0.04


    c7.resize(2,3);
    cout<<"c7=\n"<<c7<<endl;
    // Output:
    // c7=
    // 0           0           78
    // 0           0           4

    return 0;
}
```

For other useful methods for `TVec` and `TMat` and more details on their implementation, see files `PLearn/plearn/math/TVec_{decl,math}.h` and `PLearn/plearn/math/TMat_{decl,math}.h`

2.1.2 Mathematical Manipulations

Though you might want to implement certain mathematical functions or operators yourself, many mathematical manipulations for `TVec` and `TMat` are already implemented in PLearn.

In `PLearn/plearn/math/TMat_maths_impl.h`, many mathematical operators, such as `+`, `-`, `*`, `/`, `+=`, `-=`, `*=` and `/=` are already overloaded. When using `+`, `-`, `*` or `/`, a new vector/matrix is created as the result of the operation, and when using `+=`, `-=`, `*=`, `/=`, the operand on

the left is modified and no object is created. Also, many vector/matrix products are implemented. Given the vector x and y and the matrices A, B and C:

- `dot(x,y)` computes $x'y$
- `product(y,A,x)` computes y such that $Ax = y$
- `transposeProduct(y,A,x)` computes y such that $A'x = y$
- `product(C,A,B)` computes C such that $AB = C$
- `transposeProduct(C,A,B)` computes C such that $A'B = C$
- `externalProduct(A,x,y)` computes A such that $xy' = A$

Moreover, the functions `productAcc`, `transposeProductAcc` and `externalProductAcc` perform the same operations but accumulate the result of the computations in the modified data structure instead of overwriting what it initially contained. For example, the computation of $Ax + Ay = z$ can be done by the following calls: `product(z,A,x)` followed by `productAcc(z,A,y)`.

Many other standard functions can be found in `PLearn/plearn/math/TMat_maths_impl.h`. The most popular are probably `sign`, `max`, `argmax`, `min`, `argmin`, `softmax`, `exp`, `abs`, `log`, `logadd`, `sqrt`, `sigmoid` and `tanh`.

When considering to implement a given mathematical function on vectors and matrices in PLearn, some time can be saved by first looking in `PLearn/plearn/math/TMat_maths_impl.h` in order to verify whether it has already been implemented.

In `PLearn/plearn/math/TMat_maths_specialisation.h`, optimized versions of vector/matrix operators for specific data types and relying on the BLAS library can be found. Also, in `PLearn/plearn/math/plapack.h`, other specialized functions for vectors and matrices (matrix inverse, eigenvalue and singular value decomposition, linear system solver, etc.) relying on the LAPACK library can also be found.

2.1.3 Loading and saving

You can load and save a Mat with the following code (VMat.h must be included):

```
#include <plearn/math/TMat_maths.h>
#include <plearn/var/Var_all.h>
#include <plearn/vmat/VMat.h>
#include <plearn/db/getDataSet.h>

using namespace PLearn;

int main(int argc, char** argv)
{
```

```
    Mat c(3,2);
    c(1,1)=1.0;
    c(1,0)=4.0;
    c(2,0)=5.0;
    c(0,1)=73.0;
    c(0,0)=78.0;
    c(2,1)=5.0;

    // save into a pmat file
    c.save("save.pmat");

    // save into an amat file
    VMat vm(c);
    vm->saveAMAT("save.amat");

    // load from a file
    VMat vm2 = getDataSet("save.pmat");
        // it could have been "save.amat"
    Mat m = vm2.toMat();
    cout<<m;

    return 0;
}
```

2.2 How to create a PLearner?

PLearner is the super class for learner. Here we describe how to create a PLearner. PLearner is a subclass of Object so if you want to know more about what you are doing, go to section 3.2.

2.2.1 What?

A `PLearner` is an object intended to *learn* some structure in the data that is provided to it during a *training phase*, and use this knowledge to do some inference on (usually new) data during a *testing phase*.

A typical training phase includes:

- setting some options to control the behaviour during learning;
- providing the data the algorithm will learn from. This data might include *targets* if the goal is to perform classification or regression for example (but not for density estimation), and can include per-sample *weights*;
- calling the method `train()`, that performs the actual learning.

At this point, the PLearner is ready to be used on *test* data. You can:

- compute an output value from a given test input (a trained PLearner can be used to process data);
- compute an output value and a cost from a given test input and expected target. This can be useful to test the error of the algorithm on new data, but a cost can as well be a measure of uncertainty or a reconstruction error;
- perform this last operation on a whole dataset and accumulate statistics.

2.2.2 Where?

If your learner is experimental, and at least until it compiles and work perfectly under every situation, you should not commit it in `$PLEARNDIR/plearn_learners` with the other ones, but it is still a good idea to put it in the version tracking system, and to commit often. When your learner works robustly, you can move it into the corresponding subdirectory of `plearn_learners`.

If you have an account on LisaPLearn, the best is to use `$LISAPLEARNDIR/UserExp/your_login/any_path`, if you don't you can create a subdirectory in `$PLEARNDIR/plearn_learners_experimental` and use it as a working directory.

2.2.3 How?

Here is a step-by-step example of how to implement `MyLearner`:

1. Once you are in your working directory, type:

   ```
   $ pyskeleton PLearner MyLearner
   ```

   where `MyLearner` is the name of the PLearner you want to create.

   This will create two files, `MyLearner.h` and `MyLearner.cc`, from a template. These files contain the prototypes of the methods you need to implement in order to follow the PLearner interface, and some comments to help you filling them.

2. Edit `MyLearner.h`, and add (possibly short) Doxygen documentation about what your learner is supposed to do. Something like:

   ```
   namespace PLearn {

   /**
    * Learns the meaning of life.
    * This class learns how to find the meaning of life through the application
    * of stochastic methods. Tests can be performed on several 42-dimensions
    * vectors.
    *
   ```

```
 * @todo Make God fit into this framework.
 *
 */
class MyLearner : public PLearner
```

3. Declare your public options. These will typically be the hyperparameters of your algorithm, and the options allowing to switch between different methods. These are the options the user will need to provide for your algorithm to know what to do, but they can change during the learning phase. Don't forget to put comments:

```
class MyLearner : public PLearner
{
    typedef PLearner inherited;

public:
    //#####  Public Build Options  ############################################

    //! ### declare public option fields (such as build options) here
    //! Start your comments with Doxygen-compatible comments such as //!

    //! Initial parameters, specified by the user
    Vec init_params;

    /**
     * Method to use for performing learning.
     * One of:
     *   - "none": use raw data
     *   - "first": first method
     *   - "second": second method
     */
    string learning_method;
```

You can skip the "public methods" section for the moment.

4. Declare your protected options. These will typically be parameters learned from your data, or from the public options (cached to avoid always accessing them).

```
protected:
    //#####  Protected Options  ###############################################

    // ### Declare protected option fields (such as learned parameters) here

    //! Number of initial parameters
    int nparams;

    //! 'learning_method' as number: 0 for none, 1 for first, 2 for second
    int method;
```

```
    //! Learned parameters: a matrix of size (nparams * inputsize())
    // (inputsize() is a method of PLearner)
    Mat learned_params;
```

5. Declare other variables you will need during learning or computations, and you don't want to reallocate each time. This members can be protected or private, depending if your subclasses are likely to use them.

```
    //####  Not Options  ######################################################
    //! Stores intermediate results
    Vec tmp;
```

6. Edit `MyLearner.cc`. First, fill the PLearn documentation of the class. This is usually the same as the doxygen one.

```
namespace PLearn {
using namespace std;

PLEARN_IMPLEMENT_OBJECT(
    MyLearner,
    "Learns the meaning of life.",
    "This class learns how to find the meaning of life through the"
    " application\n"
    "of stochastic methods. Tests can be performed on several 42-dimensions\n"
    "vectors.\n");
```

7. Write the default constructor. First, initialize all fields which need it (such as int, bool, real...), and the ones you want to, to a default value. You can skip some (like the Vec and Mat, that have a reasonable default constructor), but you have to initialize the fields in the same order you declared them.

```
MyLearner::MyLearner()
    : learning_method("none"),
      nparams(-1),
      method(0),
      tmp(42)
{
}
```

   The comment says you may want to call `build_()` to finish the construction of the object. For this default constructor, you probably don't want to do it, because `build_()` will be called anyway after setting the actual option values.

8. *[Optional]* Write other constructors. You can write new constructors, that would take (for example) as arguments all the parameters needed to build the learner completely. In such a constructor, you may want to call `build_()` (so you are sure everything is usable right after construction) or build everything

```
// In MyLearner.h
MyLearner( Vec the_init_params, string the_learning_method = "none" );

// In MyLearner.cc
// If everything is in the constructor:
MyLearner::MyLearner( Vec the_init_params, string the_learning_method );
    : init_params(the_init_params),
      learning_method(the_learning_method),
      nparams(the_init_params.length()),
      method(-1),
      learned_params(nparams, max(0,inputsize())),
      tmp(42)
{
    learning_method = lowerstring(learning_method);
    if( learning_method == "none" )
        method = 0;
    else if( learning_method == "first" )
        method = 1;
    else if( learning_method == "second" )
        method = 2;
    else
        PLERROR("MyLearner - learning_method '%s' is unknown.",
                learning_method.c_str());
}

// If we prefer to call build():
MyLearner::MyLearner( Vec the_init_params, string the_learning_method );
    : init_params(the_init_params),
      learning_method(the_learning_method),
      nparams(-1), method(-1)
{
    // We are not sure inherited::build() has been called, so:
    build();
}
```

9. Now, declare (in the sense of PLearn) the options of the learner, as you do for any Object. The options we had in section "Public Build Option" will be labeled `buildoption`, the ones in "Protected Option" will be labeled `learntoption`, and the ones in "Not Options" will *not* be declared.

```
void MyLearner::declareOptions(OptionList& ol)
{
    // ### Declare all of this object's options here.
    // ### For the "flags" of each option, you should typically specify
    // ### one of OptionBase::buildoption, OptionBase::learntoption or
    // ### OptionBase::tuningoption. If you don't provide one of these three,
    // ### this option will be ignored when loading values from a script.
```

```
    // ### You can also combine flags, for example with OptionBase::nosave:
    // ### (OptionBase::buildoption | OptionBase::nosave)

    // First, the public build options
    declareOption(ol, "init_params", &MyLearner::init_params,
                  OptionBase::buildoption,
                  "Initial parameters");

    declareOption(ol, "learning_method", &MyLearner::learning_method,
                  OptionBase::buildoption,
                  "Method to use for performing learning.\n"
                  "One of:\n"
                  "  - "none": use raw data\n"
                  "  - "first": first method\n"
                  "  - "second": second method\n");

    // Then, the learned options
    declareOption(ol, "nparams", &MyLearner::nparams,
                  OptionBase::learntoption,
                  "Number of initial parameters");

    declareOption(ol, "method", &MyLearner::method,
                  OptionBase::learntoption,
                  "'learning_method' as a number:\n"
                  "0 for none, 1 for first, 2 for second.\n");

    declareOption(ol, "learned_params", &MyLearner::learned_params,
                  OptionBase::learntoption,
                  "Learned parameters: a matrix of size (nparams *"
                  " inputsize())");

    // Now call the parent class' declareOptions
    inherited::declareOptions(ol);
}
```

10. Now, if you include your header in `plearn_inc.h`:

    ```
    #include <plearn_learners_experimental/some_path/MyLearner.h>
    ```

    you should be able to compile `plearn`, and to have help of all the "buildoption" options when typing:

    ```
    $ plearn help MyLearner
    ```

    Options that are not labeled "buildoption", such as the "learntoption" options defined above, do not appear: it is not necessary to provide them (it could even confuse your learner if their values are inconsistent).

11. Now, let's implement a basic version of the `build_()` method. It is intended to let you test (and debug) your class in a few easy situations, but you will have to rewrite a more complete version later (see section 2.2.5), so that it works correctly in every case.

    The goal of `build()` is to ensure that the object is in a consistent state, and ready to be used. This method calls `inherited::build()` (in our case, `PLearner::build()`), and then `build_()`, which we have to implement. It is called in various situations, so a correct version of `build_()` should check everything. Now, we will only focus one simple scenario, where the sequence of methods called is:

    - `MyLearner()`,
    - `build()`,
    - `setOption(...)` possibly on every build option,
    - `build()`,
    - `setTrainingSet( some_trainset )`,
    - `build()`.

    The first call to `build_()` can be used to do some initializations that do not fit in the default constructor, or thar use the default values of parent object (PLearner), for example. The first and second calls set the values of the parameters learned from build options. Here, we resize `learned_params` only if the parameter `inputsize_` is positive (meaning the input size of the learner have been set).

```
//! @todo rewrite this method to work in every case
void MyLearner::build_()
{
    // ### This method should do the real building of the object,
    // ### according to set 'options', in *any* situation.
    // ### Typical situations include:
    // ###  - Initial building of an object from a few user-specified options
    // ###  - Building of a "reloaded" object: i.e. from the complete set of
    // ###    all serialised options.
    // ###  - Updating or "re-building" of an object after a few "tuning"
    // ###    options have been modified.
    // ### You should assume that the parent class' build_() has already been
    // ### called.

    nparams = init_params.length();

    if( inputsize_ > 0 )
        learned_params.resize(nparams, inputsize());

    learning_method = lowerstring(learning_method);
    if( learning_method == "none" )
        method = 0;
    else if( learning_method == "first" )
```

```
        method = 1;
    else if( learning_method == "second" )
        method = 2;
    else
        PLERROR("MyLearner - learning_method '%s' is unknown.",
                learning_method.c_str());
}
```

The member `PLearner::inputsize_` is equal to the size of the elements the learner takes as input, or -1 if they are not set (or variable). There are two possibilities to have it set: calling `setTrainingSet(...)` on some `VMat` (see section 2.2.7), in that case it will be set to the inputsize of that `VMat`, or having it set as an option (or read from a script).

In the function above, if `inputsize_` is set, no matter how, it will resize `learned_params`, and that is what we want: always use all the informations available.

12. Since PLearn uses smart pointers (see section 3.1.2), when we make a copy of an Object, it is by default a "shallow" copy, meaning that the pointers are copied, but still point to the same actual data. Each Object (hence each PLearner) class has to implement a method called `makeDeepCopyFromShallowCopy` that creates a real, "deep" copy of every field we have a pointer to (and recursively).

    In this step, we ensure that every member of our PLearner that is (or contain) a smart pointer (a `PP<something>`) will be "deepCopied". Usually, it concerns the members of type `TVec<something>`, `Vec`, `TMat<something>`, `Mat` and of course `PP<something>`. If you use classes defined elsewhere, be careful that a class name could be a `typedef` to `PP<something>`: for instance a `VMat` is a PP¡VMatrix¿, so you would have to call `deepCopy` on it.

    ```
    void MyLearner::makeDeepCopyFromShallowCopy(CopiesMap& copies)
    {
        inherited::makeDeepCopyFromShallowCopy(copies);

        // ### Call deepCopyField on all "pointer-like" fields
        // ### that you wish to be deepCopied rather than
        // ### shallow-copied.
        // ### ex:
        // deepCopyField(trainvec, copies);

        deepCopyField(init_params, copies);
        deepCopyField(learned_params, copies);
        deepCopyField(tmp, copies);
    }
    ```

    Don't forget to remove these lines when finished:

    ```
        // ### Remove this line when you have fully implemented this method.
        PLERROR("StackedModulesLearner::makeDeepCopyFromShallowCopy not fully (correctly) impl
    ```

13. We will now implement the methods that are specific to PLearner. Let's begin with `outputsize()`. Whereas the value of `inputsize()`, `targetsize()` and `weightsize()` will be automatically set from the training set's sizes (see 2.2.7), you have to define the output size of your learner. It can depend on `inputsize()` and other parameters, but you should not change it during the learning or testing phase.

```
int MyLearner::outputsize() const
{
    // Compute and return the size of this learner's output (which typically
    // may depend on its inputsize(), targetsize() and set options).
    if( method == 0 )
        return 42;
    else if( method == 1 )
        return inputsize()+1;
    else if( method == 2 )
        return inputsize()*2;
    else
    {
        PLERROR("MyLearner::outputsize() - method '%i' is unknown.\n"
                "Did you call 'build()'?\n", method);
        return 0; // to avoid warning, we must return in every case
    }
}
```

14. The method `forget()` is used to forget everything the PLearner learned during the training phase. It can be called when changing the training set (for the first training set, or if what it learned with one data set is not usable with another one), or if we want to try to learn with different parameters, for example. The "learntoption" parameters we have set during `build_()` are not affected.

```
void MyLearner::forget()
{
    //! (Re-)initialize the PLearner in its fresh state (that state may depend
    //! on the 'seed' option) and sets 'stage' back to 0 (this is the stage of
    //! a fresh learner!)
    /*!
      A typical forget() method should do the following:
      - initialize a random number generator with the seed option
      - initialize the learner's parameters, using this random generator
      - stage = 0
    */
    learned_params.clear();
    stage = 0;
}
```

15. The method `computeOutput(input, output)` computes an *output* vector from the *input* part of a data point. It is routinely called when the PLearner is trained (because

it is what you train it for!), but can also be used during training. The implementation really depends on what you want your learner to do.

```
void MyLearner::computeOutput(const Vec& input, Vec& output) const
{
    // Compute the output from the input.
    int nout = outputsize();
    output.resize(nout);

    if( method == 0 )
    {
        tmp += sum(input);
        output << tmp;
    }
    else if( method == 1 )
    {
        output.subVec(0,input.length()) << input;
        output[ inputsize() ] = sum(tmp);
    }
    else if( method == 2 )
    {
        if( inputsize() != 42 )
            PLERROR("MyLearner::computeOutput: inputsize() is '%d', but\n"
                    "Learning method 'second' only works when inputsize() =="
                    " 42.\n", inputsize());
        output.subVec(0,42) << input;
        output.subVec(42,42) << tmp;
    }
}
```

16. The method `computeCostsFromOutput(...)` is usually called after `computeOutput`, because it computes the costs from *already* computed outputs, knowing the actual target (if any). Note that you can have several costs, each of one being a scalar. Typical costs are squared distances between output and target, NLL, hinge loss... It mainly depends on how you measure the performance of your learning algorithm.

    Do not forget the "const" at the end of the declaration line.

```
void MyLearner::computeCostsFromOutputs(const Vec& input, const Vec& output,
                                        const Vec& target, Vec& costs) const
{
    // Compute the costs from *already* computed output.
    costs.resize(1);
    costs[0] = powdistance(output, target, 2); // squared error
}
```

17. *[Optional]* The method `computeOutputAndCosts(...)`, as it names tells, computes both the output vector and the costs, from an input vector and the corresponding

target. There is a default implementation in the parent `PLearner` class, that calls `computeOutput` and then `computeCostsFromOutput`, but you may want to reimplement it to improve efficiency.

It would be the case if computing the output gives you the costs as well (if you don't need the target). Then, you may want to implement `computeOutputAndCosts`, and make `computeOutput` and `computeCostsFromOutput` call it.

You may also want to reimplement the method `computeCostsOnly`.

18. The method `train()` performs the actual training. Its implementation will mostly depend on your algorithm. The pseudo-code included in `MyLearner.cc` will give you an idea of the general structure of a `train()` method, but it is possible that your learner doesn't really fit in this.

    Vague advice: don't hesitate to add helper functions or routines to prevent `train()` from being too big an monolithic; keeping statistics during training can be useful; you can use `computeOutput`, `computeOutputAndCosts` and `computeCostsFromOutput` from inside `train()` to avoid code duplication (but it might be impossible for some learners).

```
void MyLearner::train()
{
    // The role of the train method is to bring the learner up to
    // stage==nstages, updating train_stats with training costs measured
    // on-line in the process.

    static Vec input;  // static so we don't reallocate memory each time...
    static Vec target; // (but be careful that static means shared!)
    static Vec train_costs;

    input.resize(inputsize());    // the train_set's inputsize()
    target.resize(targetsize());  // the train_set's targetsize()
    real weight;

    // This generic PLearner method does a number of standard stuff useful for
    // (almost) any learner, and return 'false' if no training should take
    // place. See PLearner.h for more details.
    if (!initTrain())
        return;

    // learn until we arrive at desired stage
    for( ; stage < nstages ; stage ++ )
    {
        // clear stats of previous epoch
        train_stats->forget();

        // loop over all examples in train set
        for( int sample=0 ; sample<nsamples ; sample++ )
```

```
            {
                train_set->getExample(sample, input, target, weight);
                computeOutputAndCosts(input, target, output, train_costs);
                // keep statistics of costs
                train_stats->update(train_costs);

                // minimize the cost on current sample, modifying learned_params
                // this function is defined elsewhere in this file...
                minimizeByVariationalMethods(input, target, output, train_costs);
            }
            train_stats->finalize(); // finalize statistics for this epoch
        }
    }
```

19. `getTestCostNames()` and `getTrainCostNames()` return, as a vector of strings, the names of the costs computed during testing and during training (respectively). These values are used to interpret the accumulated statistics.

    `getTrainCostNames.length()` should be equal to the `train_costs` vector used to update the training statistics, and `getTestCostNames.length()` should be equal to the length of the `costs` vectors computed in `computeCostsFromOutput`. However, the train and test costs are not necessarily the same, and do not have necessarily the same length.

    ```
    TVec<string> MyLearner::getTestCostNames() const
    {
        return TVec<string>(1, "squared_distance");
    }

    TVec<string> MyLearner::getTrainCostNames() const
    {
        return getTestCostNames();
    }
    ```

2.2.4 And now?

Now, you can compile `plearn`, and try your learner in different situations, from a program as well as from a script, with different training sets, debug it, and see if the results seem normal. It is usually a good thing to try it with a simple "toy" dataset, for which you can do the computation by hand, in order to be sure everything works as intended.

2.2.5 A build_() that works in every situation

1. split the building of the different parts of your PLearner into different functions,

2. call one of these functions as soon as you have all the elements needed,

3. it is better to compute twice the same thing at build time and lose a bit of efficiency, than to think that everything has already been set to the right values when one parameter has changed.

2.2.6 Useful members and methods defined in PLearner class

1. `inputsize()`
2. `setTrainingSet()`
3. `train_set`
4. `train_stats`
5. `stage`
6. `nstages`

2.2.7 Datasets

In PLearn, the data structure for datasets (training, validation and test sets) correspond to the `VMatrix` class. Conceptually, it corresponds to a matrix where each row is a sample (training or test example). The length of the `VMatrix`, that is the number of samples it contains, is given by the `length()` method.

A row is divided in three parts called the input, target and weight parts. Their respective sizes are given by the methods `inputsize()`, `targetsize()` and `weightsize()` of the `VMatrix` object. The total width of the `VMatrix` is given by the method `width()` and should be equal to the sum of the input, target and weight part sizes. The weight size should also be either 1 or 0, that is a sample either has a weight or does not.

There are two ways of accessing a sample of a `VMatrix`. The method `getRow(i, row)` can be called, where `i` is the index of the row and `row` is a `Vec` which will be filled with the input, target and weight parts of the i^{th} sample. The length of `row` should already be set to the width of the VMatrix.

Another possibility is to call `getExample(i, input, target, weight)`, where `input` and `target` are `Vec` objects and will be filled with the input and target parts of the i^{th} sample. The `weight` variable is a reference to a `real`, which will be equal to the weight of the i^{th} sample. The `input` and `target` vectors do not have to be sized according to the `VMatrix` input and target part sizes. They will be resized appropriately by `VMatrix` (this is possible since they are passed as references to the method).

Usually, instead of manipulating `VMatrix` objects, you will have access to a `VMat` object, which can be seen as a pointer to a `VMatrix` object and should be used as such. The `VMat` class inherits from the class `PP<VMatrix>`, that is the class of smart pointers for `VMatrix` objects. To know more about smart pointers, see section 3.1.2.

The `VMatrix` object is implemented in files `PLearn/plearn/vmat/VMatrix.{cc,h}`, where you will find many more methods for this class. However, the ones that we described in this section will usually be sufficient for the implementation of a `PLearner`.

2.2.8 Testing phase

TODO

2.2.9 How to get the dataset?

TODO

2.2.10 How to manage the dataset?

TODO

2.2.11 If you need gradients on a cost function...

Go read the section 3.3 and the section 4.3.

Chapter 3

Intermediate

3.1 Low-level concepts

3.1.1 Important compilation flags

TODO: explication

- BOUNDCHECK or nothing
- USEFLOAT or USEDOUBLE
- LITTLEENDIAN or BIGENDIAN

Default with Pymake and Linux is BOUNDCHECK, USEDOUBLE, LITTLEENDIAN.

3.1.2 Smart Pointers

Memory management is one of the most error-prone aspects of traditional C and C++ programming. PLearn makes it easier through the use of *reference-counted smart pointers*.

Traditionally, there are two basic ways an object can typically be created:

- on the stack:

```
void f()
{   // Beginning of scope
    MyClass myinstance;
        // memory is allocated on the stack,
        // and constructor is called

    myinstance.dosomething();
        // methods and members are called using a dot
```

```
    }   // when exiting the scope, destructor of object is
        // called automatically and stack memory is freed
```

- by calling new:

```
    void f()
    {   // Beginning of scope

        MyClass* ptr = new MyClass();
            // memory is allocated on the heap by the "new"
            // opearator, which returns a pointer

        ptr->dosomething();
            // methods and members are called using "->"

        delete ptr;
            // we have to call "delete" explicitly,
            // because object is NOT automatically destroyed
    }       // when leaving the scope
```

In more complex cases, where several ojects may contain pointers to other objects, keeping track of when to delete an object quickly becomes a complex and error-prone bookkeeping task.

PLearn uses reference-counted smart pointers to automate this, so that you don't have to worry about calling delete. It is based on a *smart pointer* template (PP which stands for PLearnPointer) that can be used on any class that derives from SmartPointable. A SmartPointable object contains the count of the number of smart pointers that point to it, and is automatically destroyed when this *reference count* becomes 0 (i.e. when nobody any longer points to it)

```
class MyClass: public SmartPointable;

void f()
{   // Beginning of first scope
    PP<MyClass> ptr = new MyClass();
        // memory is allocated on the heap
        // (reference count for object is 1)

    { // Beginning of second scope
        PP<MyClass> ptr2 = ptr;
            // ptr2 and ptr point to the same object
            // (reference count becomes 2)

    } // Object is not destroyed upon exiting the second scope
      // (reference count becomes 1)
```

```
    ptr->dosomething();
        // methods and members are called using "->"

} // Object is automatically destroyed here
  // when reference count becomes 0
```

It is possible to mix traditional pointers to an object with smart pointers, and there are automatic conversions between the two. However, in general we discourage doing this, although it might prove useful in some situations (such as to keep a pointer to the actual specific type of the object rather than its base-class). If you do mix them, just remember that the object will get deleted as soon as the last smart pointer pointing to it is gone (when it gets out of scope for instance), regardless whether there are still traditional pointers pointing to it (the automatic reference count can only counts smart pointers!).

Many base classes in PLearn have an associated smart pointer type with a similar (and usually shorter) name, as shown in the following table. Sometimes this corresponding smart pointer type is a simple typedef to the type PP¡*baseclass*¿.

But we also often specialised them (by deriving PP¡*baseclass*¿) to add operators and methods for user convenience. So that, for instance, element at row i and column j of a VMat m can be accessed as m(i,j) as an alternative to the more verbose m-¿get(i,j).

base class	smart pointer
VMatrix	VMat
Variable	Var
RandomVariable	RandomVar
Kernel	Ker
CostFunction	CostFunc
StatsIterator	StatsIt

Several concepts in PLearn can be seen as having two levels of implementation:

1. A base class and its derived classes form the basic internal working mechanism for the concept which can be extended by deriving new classes. We call this the *designer level*.

2. A correponding smart pointer type for the base class, and a number of utility functions give a more user-friendly syntax to use the concept. They are mostly wrapping and syntactic sugar around the *designer level* classes. We call this the *user level*.

The person who only wishes to use the library typically doesn't need to understand all the details of the designer level hierarchy. Some concepts (such as Var) can be manipulated almost entirely through the smart pointer type and user-level functions, although knowing the most useful methods of the uinderlying base class, and the role of each subclass certainly doesn't harm.

3.2 How to subclass a PLearn Object

3.2.1 Object

PLearn defines an **Object** class. There is not much to it. Its role is mainly to standardise the methods for printing, saving to file, and duplicating objects. Not all classes in PLearn are derived from Object (many low-level classes aren't). But all non-strictly-concrete classes (i.e. those with virtual methods) in PLearn derive from Object. This includes the Learner base class.

Object allows an easy support for a number of useful generic facilities:

- automatic memory management (through reference counted smart pointers: **Object** derives from **PPointable**)
- serialization/persistence (`read, write, save, load`)
- runtime type information (`classname`)
- displaying (`info, print`)
- deep copying (`deepCopy`)
- a generic way of setting options (`setOption`) and a generic `build()` method (the combination of the two allows for instance to change the object structure and rebuild it at runtime)

3.2.2 Creating a basic class deriving from Object

First, you can use `pyskeleton`, a python script which creates automatically the .h and .cc files.

`pyskeleton Object Person` creates a class called Person derived from Object.

The first thing to do is to fill the .h file.

Example:

```
...

private:
    typedef Object inherited;

protected:
    // *********************
    // * protected options *
    // *********************

    // ### declare protected option fields
    // ###  (such as learnt parameters) here
```

```
    // ...

public:
    // here we had the good things
    string firstname;
    int age;
```

Then you just have to fill the `declareOptions` method in the .cc .

```
void Person::declareOptions(OptionList& ol)
{
    // ### Declare all of this object's options here.
    // ### For the "flags" of each option, you should typically specify
    // ### one of OptionBase::buildoption, OptionBase::learntoption or
    // ### OptionBase::tuningoption. If you don't provide one of these three,
    // ### this option will be ignored when loading values from a script.
    // ### You can also combine flags, for example with OptionBase::nosave:
    // ### (OptionBase::buildoption | OptionBase::nosave)

    // ### ex:
    declareOption(ol, "firstname", &Person::firstname,
                  OptionBase::buildoption,
                  "Help text describing this option");

    declareOption(ol, "age", &Person::age,
                  OptionBase::buildoption,
                  "Help text describing this option");
// ...

  // Now call the parent class' declareOptions
  inherited::declareOptions(ol);
}
```

3.2.3 Setting option fields and calling build()

There are several techniques to implement the facilities of finishing building afterwards and named parameters. In PLearn, we typically use public option fields (or sometimes protected fields with setter methods) and a public `build()` method that does the actual building. Think of those public fields as really nothing but named constructor parameters, and `build()` as the one and only true constructor.

The building of *me* in the previous example could then look as follows:

```
#include "Person.h"

using namespace PLearn;
```

```
int main(int argc, char** argv)
{
    Person me; // default constructor can set default values
               // for the option-fields
               // for ex: suppose default profession is "student"
    me.firstname = "Pascal";
    me.age = 29;
    me.build(); // finalize the building process

    cout<<me.firstname;
}
```

Note that there has to be a default (empty) constructor, whose role is also to set the default values of the parameters.

3.2.4 A generic way of setting options from "outside"

Sometimes, you want to set options and build an object from some form of interpreted language environment or from a text description, etc. That is to say from "outside" a C++ program. For this, *PLearn* provides the setOption method. Suppose `Person` is a subclass of **Object**, then we could do the following:

```
#include <plearn/base/Object.h>
#include "Person.h"

using namespace PLearn;

int main(int argc, char** argv)
{
    Object *o = new Person(); // o is a smart pointer to an
                              // object whose true type is Person
    o->setOption("firstname","Pascal");
    o->setOption("age","29");

    Person *p = dynamic_cast<Person*>(o);

    cout<<p->firstname;
}
```

Note that `setOption` takes 2 strings: the name of the option, and its value serialised in string form. Strings are universal because anything can be represented (serialized) as a string. Actually, setOption calls a lower-level method called `readOptionVal` which reads the option value from a stream (a string stream in this case...) rather than a string. Similarly there is a `getOption` method which returns a string representation of a named option, and whose implementation simply calls `writeOptionVal` on a string stream.

3.2.5 Building an object from its specification in a file

Building an object from a specification in a file is a natural extension of the setOption/build mechanism. Suppose we now have a file `me.psave` containing the following text:

```
Person( firstname="Pascal";
        age = 29;
      );
```

In the following code, we a way to build `me` from its description in the file.

```
#include <plearn/base/Object.h>
#include "Person.h"

using namespace PLearn;

int main(int argc, char** argv)
{

    Object* o = loadObject("me.psave");
    Person *p = dynamic_cast<Person*>(o);

    cout<<p->firstname;

    return 0;
}
```

There are others ways to do that:

```
string filename = "me.psave";

// 1) The loadObject function
{
    Object *me;
    me = loadObject(filename);
}

// 2) What loadObject actually does
{
    ifstream in(filename.c_str());
    Object *me = readObject(in);
}

// 3) An alternative (loadObject actually calls Object::read)
{
    Person me;
```

```
    ifstream in(filename.c_str());
    me.read(in);
}

// 4) An alternative using the global generic
//    plearn::read function
{
    Object *me;
    ifstream in(filename.c_str());
    ::read(in, me);
}

// 5) What if we have the string representation at hand?
{
    // get the content of the file as a string
    // (function in fileutils.h)
    string description = loadFileAsString(filename);
    Object *me = newObject(description);
}
```

Naturally, all that these functions do is parse the description in the file, and call `readOptionVal` (the lower-level equivalent of `setOption`) for each specified option, before finally calling `build()`.

Note that options may have arbitrarily complex types. They are not limited to strings and numbers; in particular they may themselves be compex objects or arrays of things. For example:

```
Drawing(
    color = "blue";

    # path is an array of objects
    path = [ Line(x0=0, y0=0, x1=10, y1=20);
             Line(x0=0, y0=0, x1=10, y1=20, width=2);
             Circle(x=20; y=30; radius=5.3, fill=true);
           ];
    );
```

Finally, you should use a SmartPointable for Person, as seen before in 3.1.2.

```
#include <plearn/base/Object.h>
#include "Person.h"

using namespace PLearn;

int main(int argc, char** argv)
{
```

```
    Object* o = loadObject("me.psave");
    PP<Person> p = dynamic_cast<Person*>(o);

    cout<<p->firstname;

    return 0;
}
```

3.2.6 Human description versus saved object

The `me.psave` file in the previous section may have been produced either manually by a human being, or automatically by calling

```
plearn::save("me.psave",me);
```

on a previously constructed `Person me` object.

The mechanism for building an object is the same in both cases: it automatically calls a series of `readOptionVal` followed by `build()`. However the options specified in both cases are not always the same:

- A hand-written description file will typically be used to give a small number of options for the *initial building* of an object (with the other options taking their default value).

- A file resulting from a saved object, will typically include *everything* that is necessary to reconstruct a new instance in the full and exact same state as the instance that was saved. This may include options, such as the learnt synaptic weights of a neural network, that are not given at the time of *initial building*, but only when *reloading* a serialised object.

We call the options typically used for initial building **build options**, and the second type **learnt options**. Note that the behaviour of the `build` method may have to be quite different when we are *reloading* a saved object (and providing it with *learnt options*) from when we are only doing an *initial building* (and providing it only with *build options*). It is natural that our "one and only" constructor may have to behave differently depending on the parameters it is given, but it is important to keep in mind the distinction between *build options* on one hand, and *learnt options* that are only present whe reloading, on the other hand.

There is a third conceptual category of options, that we call **tuning options**, which are used mostly to *tune* the object *after* an intial building. They often overlap with *build options*, but not necessarily, the distinction is nevertheless more conceptual than real.

3.3 Matrix-Vectors Operations with Gradients

3.3.1 Introduction to Var

The class Var is at the heart of PLearn and aims at providing matrix-variables in the mathematical sense. It is built on top of the Mat class, that provides matrix-variables in the more traditional sense of sequential computer languages.

Var should be used for Matrix-Vectors operations when you need gradients on operations. Otherwise, use Mat and Vec classes.

NO NUMERICAL COMPUTATION IS DONE AT THIS LEVEL. The purpose of the Var definitions is only to build the symbolic relationship between mathematical variables.

One can write arbitarily complex expressions using many implicit or explicit intermediary variables, and predefined functions such as in: $w = exp(-(abs(sqrt(lambda) * v)/3.0))$. This will construct an internal representation, only with a larger number of intermediate nodes to represent intermediate states (variables) of the calculations.

Each Var contains two Vec fields.

One is called value and holds the current value assigned to that variable, and the other is called gradient and is used to backpropagate gradients with respect to another variable.

Every Var has an fprop() method that updates its value field according to the value field of its direct parents.

Every Var also has a bprop() method that updates the gradient field of its direct parents according to its own gradient field (backpropagation algorithm). Note that it *accumulates* gradients into its parents gradient field.

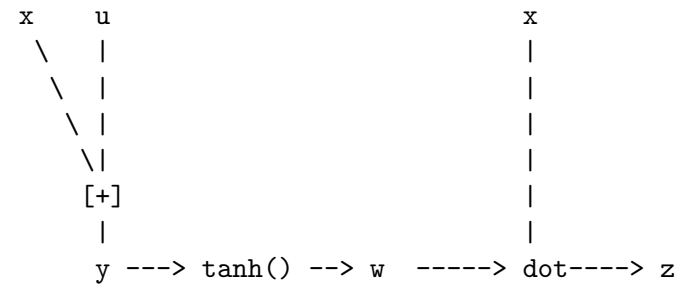
Example:

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>

using namespace PLearn;

int main(int argc, char** argv)
{
    Var v(3,2); // declares a Variable of size 3x2
    Var lambda(1,1); // declares a scalar Variable

    v->matValue(1,0)=4.0;
    v->matValue(2,0)=5.0;
    v->matValue(0,1)=73.0;
    v->matValue(0,0)=78.0;
    v->matValue(2,1)=5.0;

    cout << v->matValue << endl;
```

```
    lambda->value = 2.0;

    Var w = lambda*v;

    w->fprop();

    cout << w->matValue << endl;
    return 0;
}
```

If the expression to be calculated involves intermediate variables, fprop must be called in a correct order on all those intermediate variables before it can be called on the result variable we are interested in. For example, suppose we have $z = dot(x, tanh(y))$ where $x, u \in R^3$.

A Var builds a directed acyclic graph whose nodes are Var's, with the following structure:



To obtain the correct value of z as a function of x and u, after setting x-¿value and u-¿value, we need to perform fprop on all the intermediate nodes as well as z.

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>

using namespace PLearn;

int main(int argc, char** argv)
{
    Var x(3,1);
    Var u(3,1);

    x->matValue(0,0)=1;
    x->matValue(1,0)=2;
    x->matValue(2,0)=3;
    u->matValue(0,0)=4;
    u->matValue(1,0)=5;
    u->matValue(2,0)=6;
```

```
    cout<<x->matValue<<endl;
    cout<<u->matValue<<endl;

    Var y = x + u;
        // y is also a 3x1 matrix
    Var w = tanh(y);
    Var z = dot(x,w);
        // z is a scalar variable result of the dot product
        // of x and tanh(y)

    cout<<z->matValue<<endl;
    y->fprop();
    w->fprop();
    z->fprop();
    cout<<z->matValue<<endl;

    return 0;
}
```

To simplify the computation of values and gradients in a graph of Var's, we use a VarArray (don't forget the include).

A VarArray is simply an array of Vars, which has a method `fprop()` and a method `bprop()` which calls the `fprop()` (resp. `bprop()`) methods of all the elements of the array in the right order (note that a right order for `bprop` is the reverse of the order for `fprop`). The above function finds all the Var's on the paths from the the inputs Vars to the output Var. There are may be several input Vars so they are put in a VarArray. Once the path is obtained, we can propagate values through it with the fprop method:

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>

using namespace PLearn;

int main(int argc, char** argv)
{
    Var x(3,1);
    Var u(3,1);

    x->matValue(0,0)=1;
    x->matValue(1,0)=2;
    x->matValue(2,0)=3;
    u->matValue(0,0)=4;
    u->matValue(1,0)=5;
    u->matValue(2,0)=6;
```

```
    cout<<x->matValue<<endl;
    cout<<u->matValue<<endl;

    Var y = x + u;
        // y is also a 3x1 matrix
    Var w = tanh(y);
    Var z = dot(x,w);
        // z is a scalar variable result of the dot product
        // of x and tanh(y)

    cout<<z->matValue<<endl;

    VarArray path = propagationPath(x & u, z);
    path.fprop();

    cout<<z->matValue<<endl;
    return 0;
}
```

In the previous, the `VarArray` is useful but not essential. Let consider the following example:

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>

using namespace PLearn;

int main(int argc, char** argv)
{
    Var x(3,1);
    Var u(3,1);

    x->matValue(0,0)=1;
    x->matValue(1,0)=2;
    x->matValue(2,0)=3;
    u->matValue(0,0)=4;
    u->matValue(1,0)=5;
    u->matValue(2,0)=6;

    cout<<x->matValue<<endl;
    cout<<u->matValue<<endl;

    Var z = dot(x,tanh(x+u));
        // z is a scalar variable result of the dot product
        // of x and tanh(y)

    cout<<z->matValue<<endl;
```

```
    VarArray path = propagationPath(x & u, z);
    path.fprop();

    cout<<z->matValue<<endl;
    return 0;
}
```

This example performs exactly the same thing as the previous one. But in this case, we don't have any reference to the previous `Var y,w` to do `fprop()`. That's why a `VarArray` could be essential.

You can also use `y->fprop_from_all_sources()` instead of a `VarArray` but this reconstruct the path each time and so don't store it. It's not efficient for a multi fprop and it's not possible to back-propagate gradients.

Once we have this path, we can also back-propagate gradients. For example, if we set the gradient of z to 1,

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>

using namespace PLearn;

int main(int argc, char** argv)
{
    Var x(3,1);
    Var u(3,1);

    x->matValue(0,0)=1;
    x->matValue(1,0)=2;
    x->matValue(2,0)=3;
    u->matValue(0,0)=4;
    u->matValue(1,0)=5;
    u->matValue(2,0)=6;

    cout<<x->matValue<<endl;
    cout<<u->matValue<<endl;

    Var y = x + u;
        // y is also a 3x1 matrix
    Var w = tanh(y);
    Var z = dot(x,w);
        // z is a scalar variable result of the dot product
        // of x and tanh(y)

    cout<<z->matValue<<endl;
```

```
    VarArray path = propagationPath(x & u, z);
    path.fprop();

    cout<<z->matValue<<endl;

    z->gradient = 1.0;
    path.bprop();
    cout << "dz/dx = " << x->gradient << endl;
    cout << "dz/du = " << u->gradient << endl;

    return 0;
}
```

We obtain the partial derivatives of z with respect to x and u in their gradient field.

3.3.2 Creating

You can create Var with several methods. The main are:

```
    Var(int the_length, int width_=1);
    Var(int the_length, int the_width, const char* name);
    Var(const Mat& mat);
```

The last one id used as in the following example:

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>
using namespace PLearn;

int main(int argc, char** argv)
{
    Mat mx(3,1);
    mx(0,0) = 1;
    mx(1,0)=2;
    mx(2,0)=3;
    Var x(mx);
    cout<<x->matValue<<endl;
    return 0;
}
```

3.3.3 Manipulating

In the introduction to Var, you saw how to manipulate them.

Don't forget that all is symbolic (it will tricks you).

You can find numerous var in `PLearnplearnvar`, some are shortcuted by overloaded operators (such as +).

3.3.4 Loading and saving

Only the value

With the following method, you can load and save THE VALUE of a var (not the symbolic path).

```
#include <plearn/vmat/VMat.h>
#include <plearn/db/getDataSet.h>
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>

using namespace PLearn;

int main(int argc, char** argv)
{
    Var y(3,1);
    y->matValue(0,0)=1;
    y->matValue(1,0)=2;
    y->matValue(2,0)=3;
    cout<<y->matValue;

    // save into a pmat file
    y->matValue.save("save.pmat");

    // save into an amat file
    VMat vm(y->matValue);
    vm->saveAMAT("save.amat");


    // load from a file
    VMat vm2 = getDataSet("save.pmat");
        // it could have been "save.amat"
    Var x(vm2.toMat());
    cout<<x->matValue;
    return 0;
}
```

All the var

Var is a subclass of Object, so you can use the methods of Object as in the following example. Note that it will save all the Var, including the sub ones.

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>

using namespace PLearn;
```

```
int main(int argc, char** argv)
{
    Var x(3,1);
    Var u(3,1);

    x->matValue(0,0)=1;
    x->matValue(1,0)=2;
    x->matValue(2,0)=3;
    u->matValue(0,0)=4;
    u->matValue(1,0)=5;
    u->matValue(2,0)=6;

    cout<<x->matValue<<endl;
    cout<<u->matValue<<endl;

    Var z = dot(x,tanh(x+u));
        // z is a scalar variable result of the dot product
        // of x and tanh(y)

    cout<<z->matValue<<endl;

    VarArray path = propagationPath(x & u, z);
    path.fprop();

    cout<<z->matValue<<endl;

    save("z.psave",z);

    Object* o = loadObject("z.psave");

    // There is no PP<> nor * here,
    // because Var is already a PP<Variable>
    Var p = dynamic_cast<Variable*>(o);

    cout<<p->matValue<<endl;

    return 0;
}
```

3.3.5 Func

In order to make the usage of Var more friendly, you can use Func. The Func class is mode for those who want to make fprop on different values of Var in an elegant way. The two following examples illustrate this: they do exactly the same thing, but the first one without Func and the second one with.

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>

using namespace PLearn;

int main(int argc, char** argv)
{
    Vec a(3),c(1);
    Vec da(3),dc(1);

    // Without Func
    Var x(3,1);
    Var y = dot(x,tanh(x));
        // y is a scalar variable result of the dot product
        // of x and tanh(x)
    VarArray path = propagationPath(x,y);

    a<<"1 2.3 4";

    x->value<<a;
    path.fprop();
    c=y->value;
    cout<<a<<endl;
    cout<<c<<endl;

    a<<"4 8.3 -12";
    x->value<<a;
    path.fprop();
    c=y->value;
    cout<<a<<endl;
    cout<<c<<endl;

    dc<<2.3;
    y->gradient<<dc;
    path.bprop();
    da<<x->gradient;
    cout<<dc<<endl;
    cout<<da<<endl;

    return 0;
}
```

With Func:

```
#include <plearn/var/Var_all.h>
#include <plearn/math/TMat_maths.h>
```

```
using namespace PLearn;

int main(int argc, char** argv)
{
    Vec a(3),c(1);
    Vec da(3),dc(1);

    // With Func
    Var x(3,1);
    Var result(1,1);
    Func f(x, result ,dot(x,tanh(x)));
        // z is a scalar variable result of the dot product
        // of x and tanh(y)

    a<<"1 2.3 4";
    f->fprop(a,c);
    cout<<a<<endl;
    cout<<c<<endl;

    a<<"4 8.3 -12";
    f->fprop(a,c);
    cout<<a<<endl;
    cout<<c<<endl;

    dc<<2.3;
    f->fbprop(a,c,da,dc);
    cout<<dc<<endl;
    cout<<da<<endl;

    return 0;
}
```

3.4 Online Learning

3.4.1 OnlineLearningModule

They can be found in ${PLEARNDIR}/plearn_learners/online.

Chapter 4

Advanced

4.1 RandomVar

4.2 Function-like types

4.2.1 Ker

4.2.2 CostFunc

4.2.3 StatsIt

4.3 Optimizers

4.4 Miscalleanous utilities

The PLearnLibrary/PLearnUtils directory contains classes and functions to perform various useful things such as graphically displaying things. The Scripts directory contains a number of perl-scripts and also binary programs to both help manage the source-tree, and to manipulate matrix files.

Chapter 5

Managing software growth

A large library and collection of tools in constant development like PLearn is comparable to a vine which keeps growing branches endlessly. As developers adapt it for diverse uses, leading it simultaneously in different directions, this growth will manifest as an overwhelming tendency to:

- add new classes and files (that's fine!)
- extend existing methods of base classes by adding extra arguments (possibly with default values)
- add new member fields to existing base classes
- add new methods to existing base classes
- add new dependencies between existing files/classes (by adding new `#include` directives, due for example to the addition of a method taking a novel *type* of argument which needs to be "included").
- add dependencies to new external libraries/tools

This process is natural in the course of development, and is desirable if we want to keep the code base adaptable. However if left unchecked it will lead to its logical undesirable outcome: a huge collection of files, base classes with hundreds of member fields and hundreds of methods (many for obscure special purposes), very long compile and link times and huge executables (due to all the added dependencies) and an installation nightmare on new environments (due to the dependency on a large number of exotic "external libraries").

The aim of this chapter is to raise awareness of these issues, by shedding light on them from several angles. This will hopefully help developers get a better grip on them and take them properly into account when making design decisions (in particular decisions that imply adding a `#include` in an existing file).

5.1 A few words on the build system

Believe it or not, but the `pymake` system is designed to optimally link together *only* the object files that are *strictly necessary* for a given executable program, and among them to re-compile only those that really need re-compilation.

Now in working with PLearn, many experience the very long compilatin and linking of a huge number of files. I insist that this is not due to the build system, but only to what you, directly or indirectly, *include* in your executable.

So while it is useful for some purposes to have a `plearn` executable that includes almost everything, it is clearly not *this* `plearn.cc` that you should be compiling linking and working with when you are *developing* new algorithms to carry experiments on a more limited subject field.

Suggestion:

- Make a copy of `plearn.cc` into `mylearn.cc` and edit it to include only the few classes you (or your script) will need.
- Don't include something like `plearn_inc.h` which includes a large number of files you probably don't need. Rather copy-paste the content of `plearn_inc.h` into your `mylearn.cc` and comment out all the things you won't need.
- Don't commit *your* `mylearn.cc` under version control (so that it remains yours and seperate from the others' mylearn).

Using such a `mylearn` should make the list of files linked together and correpsonding link time a little shorter.

Now when using pymake's parallel compilation facility, the *compilation* time for plearn projects is usually reasonable. But since *linking* cannot be parallelized, link time can be problematic, especially when the object files being linked are not on your local machine (due to NFS sluggishness). So it is highly recommended, if you want link time to be acceptable, that linking does not go through NFS (i.e. link on the machine where the files are physically located).

TODO: write about `pymake -dependency`

The only remaining way to improve the issue of compilation and link time and executable size is to have a *sensible dependency graph* between files, so that unnecessary files don't get indirectly "included". This can only be achieved by developer awareness and proper restraint when the urge rises to add a `#include` in an existing file. The following section will try to give a few hints as to how this can be achieved.

Remark 1: You may have had the feeling that if only plearn was made into a proper library or set of libraries (static `.a` or dynamic `.so`) the compilation and link time problems would somehow vanish. This is simply untrue: such a system would necessarily be suboptimal compared to the **pymake** approach, as illustrated in the following example. Suppose we have a number of object files A,B,C,D,E,F,G bound together in a library, and suppose that B,C,D,E,F,G all depend on A but are otherwise independent of each other (ex: they are all direct subclasses of A). Now suppose that your executable only needs G directly. In

addition suppose, we had to make a slight modifitation in `A.h` If you go the library route, regenerating your executable will imply first regenerating the out-of-date library, which means recompiling A,B,C,D,E,F,G and rebuilding the library archive. The linking phase will then build your executable from A and G. But with the **pymake** approach, only the necessary files A and G will be recompiled (and bound together in the executable), which is a more optimal use of ressources. Having the ability to generate proper libraries may be desirable for other considerations though.

Remark 2, limitation of pymake: for efficiency reasons, pymake doesn't do a full C preprocessor pass, and thus doesn't currently understand the preprocessor logic of `#define #ifdef #if ... #else #endif`. All it understands is C and C++ comments. Thus if for ex. you `#include` something within a `#ifdef SOMEDEF`, whether SOMEDEF happens to be defined or not in the current context, pymake will still conclude there is a dependency link with the `#included` file. So in short if you want pymake to ignore a `#include` the only current way is to comment it out.

5.2 How to limit compilation and link dependencies

5.2.1 Compilation dependency versus link dependency

For the rest of our discussion, it is important to distinguish between two different kinds of dependencies:

- Saying that a file Y has a **compilation dependency** on a file X means that whenever X changes, Y is to be considered out-of-date and will have to be recompiled if we need an up-to-date version of Y. Ex: Y `#includes` X or includes another file that includes another file that includes X, etc... In short compilation dependencies affect which files will have to be "recompiled" (and thus affect re-compilation time).
- Saying that a file Y has a **link dependency** on a file X means that whenever we need to link with the corresponding object file "X.o" we will also have to link with "Y.o". In short link dependencies affect which files will have to be linked together to produce an executable (and thus directly affects link time).

These two concepts are related, yet subtly different, as will become apparent shortly.

5.2.2 How dependencies tend to creep in, and ways around them

Let us begin with a few remarks. Old-fashioed C libraries tended to put *one function* per .c file (i.e. per compilation unit). This resulted in libraries with very fine granularity, and only the *necessary functions* were included and linked in a given executable. But with classes in object oriented C++, the granularity cannot go below the class level (one class per compilation unit). And as a minimum, a class carries with it the compilation and link dependencies implied by *all its methods* and *all their argument types*.

Now when people learn C++ object-oriented programming, especially when their background programming experience is a language like Java, there is a tendency to want to

make everything a method within a class, because it appears elegant, and the OO way. But it is important to pause and reflect on the consequences and alternatives, especially when adding a method implies adding a `#include` which adds compile and link dependencies.

An example will better illustrate the alternative possible choices:

Suppose we have two independent classes A (files A.h, A.cc) and B (B.h, B.cc), and we want to add some operation `f` that requires instances of A and B as arguments (or return value). There are essentially 3 ways to implement such an operation:

- As a new method of A, taking a B as argument `A::f(B)`
- As a new method of B, taking an A as argument `B::f(A)`
- As a regular function of A and B `f(A,B)`

Now the consequences in terms of introduced file dependencies are very different.

1. Adding `A::f(B)` implies making A depend on B (i.e. you'll no longer be able to link with A without also linking with B, and any change to B.h will at least trigger a recompile of A.cc).
2. Similarly, adding `B::f(A)` implies making B depend on A.
3. On the other hand `f(A,B)` can be put in a separate file (compilation unit) possibly together with other functions of A and B. This does not create any direct compilation or link dependency between classes A and B, and the file containing `f` needs only be included (and consequently compiled and linked with) if the specific funcitonality `f` is needed.

So whenever it appears at first obvious that we need to add a method `A::f(B)` to a given class `A`, and that adding such a method forces us to add an `#include "B.h"` directive, it should trigger a red light in the mind of the developer. The red light is an invitation to consider the other two alternatives. The following considerations should then weigh in the design decision.

- Obviously, if `A::f(B)` is to be a virtual method designed to be redefined in sub-classes of A, then there is no discussing it, it should be `A::f(B)`. But if there is no reason to expect that `f` needs to be virtual in order to be redefined in sub-classes, then the alternatives can and should be considered.
- If `f` is a rarely needed functionality with a large implementation code, it is probably best left as a separate function in its own file (posibly with other similar functions).
- It may be nicer to group `f(A,B)` with other functions relating to a same topic in a separate file, rather than unreasonably increase the number of methods of A.
- But if A is meant to almost always work together with B (i.e. if it makes no sense having an A without also manipulating some kind of B), or if A already depends on B due to some other reason (such as having a member variable of type B) then `A::f(B)` is probably a reasonable design choice.

- When hesitating between `A::f(B)` and `B::f(A)` the choice should be to make the least basic, higher level, least used class depend on the most basic, lower level, more widely used class, rather than the other way round.

TODO: talk about use of forward declaration to reduce compilation dependency, but how it doesn't reduce link dependency.

5.3 Regarding external library dependencies

External dependencies tend to creep in in the codebase, and if unchecked, end up causing a nighightmare for installation, link times, and memory usage of the running software. So it is a good policy to try and limit this, based on the following three guideleines

- **If something similar already exists within PLearn, prefer using that.**
- **If it can easily be done without introducing a new dependency, then do it that way (even if it feels a little less *cool*.**
- **Prefer using an external library that is already used in PLearn (and approved) than a introducing new one.**

A few remarks regarding specific external libraries

- Use PLearn's intrusive smart pointers (`PP`, ...) for all PLearn objects. Use boost's `shared_ptr` for non PLearn classes.
- Use PLearn's PStream and serialization mechanism rather than any other C++ streams or serialization system (including std::stream).

5.4 Evolving software in a backward-compatible way

TODO: talk about

- Tolerant, evolution-friendly, serialization format
- class versioning
- new version of class (filename scheme)

Chapter 6

PLearn coding guidelines and philosophy

Several people wrote significant parts of PLearn, and if you take a closer look at the code, you will see a number of clearly different coding styles and philosophies. However, as of this writing (09/2000), the overall design and organization of most of the library is still to be blamed (or praised...) on me. So these remarks are my personal view of things, and do not necessarily reflect the opinion of everybody on the PLearn developer team, but I hope it will help you understand the reasons why things are the way they are, and hopefully have you choose to keep them that way...

Pascal

6.1 A few words on C++

Agreed, C++ *can be* a very complex language. The main reason being that it is extremely feature-rich, but that is also what makes C++ so powerful and expressive, and thus appropriate for a machine-learning library. Yet I insist on the *can be* : it doesn't always have to be, it depends a great deal on what features you choose to use and when.

People who discover C++ tend to first be overwhelmed with its wealth of features, and then seem to want to use them all at once in even the simplest piece of code (complex templates, deep multiple inheritance trees, exceptions, multiple nested namespaces; add multi-threading on top of that and you're sure to write the most unreadable, unportable and compiler-bug trigerring error-prone code ever). Finally, after great intellectual efforts, they discover that even their compiler (not to mention their debugger!) has trouble understanding it all and, if they manage to have it swallow the code, they realise that no other compiler will (portability anybody?). This is still quite true as of this writing (09/2000) and was even more so a few years ago, yet tools will keep improving until some day, hopefully, they all behave perfectly by the book, according to the standard, but until this blessed day comes, beware... Many people then give up, frustrated, and decide to go back to C, which is a shame. C++ *is* a much better language than C, especially for writing Object Oriented code, and it *does* make the programmer's life much easier... as long as *you* keep things

simple.

So please, especially if you're a beginner, keep this in mind when writing C++ code: having so many "cool" features in the language doesn't mean that you must use them all at once. Choose wisely and, if in doubt, always prefer the simplest solution...

6.2 Design goals and priorities

Any project implicitly or explicitly sets some goals and these directly influence the way code is written. With PLearn, one of the founding goal, was to be able to describe complex machine-learning experiments by assembling simple building-blocks directly in C++, without resorting to a layer of home-grown dedicated language (as experience had proven us that it is hard to grow and maintain such a language, which appears always too limited anyway). Obviously we also want to have them run efficiently (hence the choice of C++ rather than a higher level interpreted language).

Any system should ideally be simple to understand and use, lightning fast, and extremely geneal. Yet there is always a tradeoff to be made between these 3 highly desirable characteristics. Here is the priority I gave them, in the design of the library, it logically follows from the project's primary goals:

1. readability, simplicity, ease of use (and portability)
2. computational efficiency
3. genericity

6.3 Usage of C++ features in PLearn

As I mentionned earlier, moderation is good in everything, including in moderation... ;)

Function and method prototypes without parameter names

C++ code is typically divided between .h files which contain class layout, function and method prototypes, and .cc files which contain the actual implementation. Ideally, it should be possible to understand what a method or function does by looking it up only in the .h file. Comments are part of achieving this, but having a meaningful name for the parameters of the function also helps a great deal.

C++ allows you to omit parameter names in prototypes (and only give their types). This defies the purpose of clarity, and is thus considered bad practice by the author and in PLearn in general. Except for possible default values (that are to appear only in the .h), the prototype in the .h file should be identical to the definition in the .cc file and include parameter names.

(Ex: people usually have trouble understanding what float* f(float*, int, int, char, char*, float); is supposed to do, and defining a new type for each argument is *not* the right way of making this more understandable... giving them a meaningful name is.)

Basic data types

Conceptually, people usually think of 3 simple basic data types: integers, reals, and booleans (possibly 4 if you add character). C++ has them in many flavours, including signed and unsigned, several precisions, etc. These all have their use from a low-level hardware perspective (which woud have been much better if they had been given standard byte sizes by the standard...), but to the mathematically minded library-user they are an annoyance. So throughout most of PLearn, unless otherwise dictated by low-level precision or space considerations, we use only 3 types that correspond to the 3 concepts:

- **int** is used for integers
- **bool** is used for booleans
- **real** is used for reals

  `real` is defined throughout the whole library to be either `float` or `double`, depending on a compilation flag (USEDOUBLE or USEFLOAT).

Also we encourage people *not to* define a new type if it conceptually corresponds to one of those three concepts, in particular I for one (and I'm surely not alone) dislike to have to write
`namespace::subnamespace::classname::interiorclass::length_type`
when the damn thing is just an integer, if you get my point. Please use int, it saves the user keystrokes, code lookup time, and eases understanding (i.e.: genericity-- but simplicity++ and ease_of_use++, see section on desing priorities above).

The use of unsigned int types is also a source of annoyance to me, and of potential nasty bugs. Ex: `for (unsigned int i=10; i>=0; --i)`

So again, unless you really need the extra bit of precision, use int (also saves a few keystrokes).

A kind of string type is also usually seen as part of the set of basic types, but we'll discuss this in the section on the standard library.

Namespaces

Namespaces are most useful to prevent name clashes between different libraries. So ultimately, all of PLearn is to reside in the PLearn namespace. However gdb currently seems to have trouble coping with them, so the namespace directives are currently surrounded by ugly #ifdef USENAMESPACE which we usually keep undefined.

Also, for now, I do not encourage the use of sub-namespaces to organize the code within PLearn (with or without #ifdefs). It's already hard enough to get the organization right in terms of concepts, class hierarchies, and files, without introducing yet another hierarchy of things (which besides, would go mostly untested as we always compile with USENAMESPACE undefined, for now anyway).

Exceptions and runtime errors

Exceptions can be a nice and useful feature, allowing you to build sophisticated error recovery mechanisms and the like... But designing a consistent error-recovery scheme with an appropriate exception class hierarchy is a complex task. Besides in PLearn, we typically have no use for a sophisticated error recovery mechanism: a runtime error is always a sign of a bug somewhere, and the policy in PLearn is to never try to second-guess the programmer: all we want is for the program to abort immediately with a somewhat meaningful message, and the debugger to be able to trace the call. Unfortunately, as of this writing, exceptions are poorly supported by debuggers (and they can create a nightmare in multi-threaded code).

So essentially we don't use exceptions in PLearn, but a very simple runtime error mechanism: `error("my meaningful error message");` will result in a call to function `errormsg` that simply prints out the message and exits the program. Thus it is easy to set a breakpoint in `errormsg` in the debugger and trace what happened. This is a no-fuss solution that does the job. Notice that the `errormsg` function can easily be modified to throw an exception if you wish to do a proper error recovery (in case brutally exiting the program is not an acceptable behaviour).

Exceptions can also be useful for other things, but for typical runtime-errors, please use `error(`*`errormsg`*`)`.

Templates

Templates is one of the most powerful features of C++. But it's also the most complex, and the one with which compilers and debuggers have the most trouble (almost all but the simplest template code is hardly portable across compilers because of inconsistencies between them, and it was much worse a few years back!). The early versions of PLearn deliberately did not use *any* template code at all (many other librariy designers out there for whom portability was a major concern made the same choice).

As the compilers improved, I started allowing myself to use *simple* templates for things where they were *really* appropriate, (i.e. smart pointers and generic containers). And I would recommend everybody to stick to this. Please, *refrain* from using templates as much as possible: it will make your code easier to write, to read, to debug, to port, to understand, and also faster to compile. It's usually easy to later "templatize" a working and well-tested non-templated code if really needed. But it's always annoying to have to "de-templatize" a complex template code because the compiler on your new target platform cannot understand it (chances are that you won't either).

Multiple inheritance and complex class hierarchies

Multiple inheritance poses a number of technical problems and a multiple inheritance tree is also usually more difficult to understand conceptually. Therefore, PLearn uses only single inheritance and I would like to keep it that way. The only kind of "multiple inheritance" that we have is for inheriting *interfaces* (à la Java) i.e. abstract classes with only purely virtual methods.

Also we often use concrete classes, and in general prefer flat class hierarchies than very deep ones, as they are easier to comprehend.

const

const is number one on my list of C++ annoyances. But unfortunately there is no way to really do without it, so try to use it consistently, and try not to get too frustrated in case of code constipation, pardon me, const problems... there is always a (hopefully clean) way around them.

public, private, protected

There are probably too many class members that are public in PLearn. But, as we love our potential library users (they are mostly us for now anyway), we tend to avoid paranoia, and to trust them for not doing dirty things with our not-so-private members. Hell, they have access to the source code anyway!

6.4 Usage of the C++ standard library in PLearn

In early versions of PLearn, we did not use much of the standard library (as no compilers yet agreed on a standard), except for iostreams. Now that there is a well established standard, and that all compiler makers are working towards conforming to it, we are slowly moving PLearn to using more of the standard library facilities.

Strings

Many places in PLearn still use char* to represent strings, but they'll slowly be changed into using the std::string class instead. Please use string from now on. Feel free to change any usage of char* you meet into string.

A number of useful additional functions for user-friendly string manipulation can be found in file PLearnCore/stringutils.h A brief (and certainly not up to date) description of it, as well as a pointer to a quick overview of the basic string operations can be found here.

Streams

Several pieces of old PLearn code still use the C stream library (FILE* ...), but the standard C++ stream facilities is the officially approved way to go for new code.

Standard containers and algorithms

It's now OK to use STL containers wherever appropriate. Two other generic containers were previously developed for PLearn: Array is heavily used, and is a base class for a number of other specialised array types, so it is not likely to vanish any time soon (although I may

have it derive from `std::vector` one of these days). The main advantage of Array over `std::vector` is that runtime bound-checking can be turned on or off with a compilation flag (BOUNDCHECK), and there's also a user-friendly (but inefficient) syntax to build arrays from simple elements using the & operator. Hash may also be progressively abandoned in favour of std::map, hash_map (is this one part of the C++ standard?) and the like...

6.5 Naming conventions

The following naming conventions are used throughout PLearn. They are mostly inspired by the Java naming conventions. Anybody who uses or wishes to extend PLearn should be aware of them (as it makes understanding of the code easier) and try to respect them (as it will make the understanding of their code easier to other people who will have the privilege to dig into it).

To make it short and simple:

- `MyClassName`
- `myMethodName()`
- `my_variable_name` *OR* `myvariablename` *(both for member variables or otherwise)*
- `my_global_function()` *OR* `myGlobalFunction()`
- `MY_CONSTANT` *(for #define constants or other)*

Remarks:

- A classname should always start with an uppercase.
- Methods, functions, and variable names should always start with a lowercase.
- Underscores(_) should never be used in class names or method names.
- Typical methods that return a bool status should begin with is or has. Ex: `isEmpty()` `isNull()` `hasChildren()`.
- In case you want to provide a read-only accessor method to a protected or private member variable, use `varname_` for the member variable and `varname()` for the accessor method.
- The arguments of a constructor often carry initial values for member variables. We usually name the argument in the constructor 'the_varname' so that it doesn't clash with the targetted member variable (which is just 'varname')

A few reasonable exceptions are tolerated throughout the code (such as function P for probability instead of a lowercase p, or a member variable K for a kernel matrix...) But exceptions that don't serve any purpose should not be!

6.6 Final word

The PLearn library is far from perfect, it still has a lot of rough edges (my to-do list is growing every day), and there are several things that I would do differently if I was to start all over again. But it is nevertheless already a very usable tool, that for the most part, I feel, meets its primary design goals. Besides I consider good code design an iterative process: one starts with an initially rough version and iteratively refines it under the light of real-world experience. The code base is not carved in stone, it is an evolving being, and the source code is there so that you can tweak it and adapt it to your needs, and hopefully help make it better.

Chapter 7

Debugging

There are several types of problems you'll encounter, and each has a proven solving technique.

7.1 Compilation problems

Solution: learn how to program in C++

No, I mean seriously, learn C++, *thoroughly*, until you are able to truly and fully understand every single bit of the cryptic message issued by the compiler, and why on earth it may have chosen to insult your intelligence with it. Because that cryptic message always contains the solution.

In particular, you need to really understand the difference between a const thingy and a non-const thingy, because to C++ they are often two totally different beasts (although to a decent human, they may look the same at first inspection).

So make sure you truly understand *all* the subtle differences between for ex.:
`const char* MyObj::mymethod(const char* &foo, const int& bar)`
and `char* MyObj::mymethod(const char *const foo, int& bar) const`

On rare occasions, you might occasionaly stumble upon a compiler *bug* (as in *compiler internal error!!!*), in which case you may try the following: upgrade your dusty compiler to the newest less-buggy version; check on google to see if anybody else had the same problem and if they found a workaround; try to find a workaround yourself (split your call in several pieces, using intermediate variables, add a cout here, reorder the instructions there... and pray!); try posting an SOS on the appropriate newsgroups; write a bug report! Somebody somewhere, is responsible for it and might be interested in fixing the mess, or already has...

7.1.1 Frequently encountered compilation errors

To save you some time, you may look up in the following list if somebody stumbled upon the same problem.

- **typical error msg**
  explanation and fix.

7.2 Linking problems

Problems reported by the linker can have several causes:

- It doesn't find a function that's supposed to be defined somewhere in your code but isn't. A frequent case is that of instantiating an object of a class derived from a base class with a pure virtual mamber function that you forgot to define in the subclass. Another case is declaring a function in the .h and forgetting to implement it in the .cc, or (more often) implementing it with a slightly different signature (forgot Classname:: ?, forgot to put a const somewhere?)
- It doesn't find symbols because it doesn't find the required libraries. Examine the linking command, are all the necessary libraries there? Are they indeed located where you say they are? There should be no space between the -L and the library path.
- Libraries are specified in an inappropriate order. A library that depends on another library should appear before it in the linking command; the most basic libraries should be last.

7.3 Clean runtime errors

What I mean by clean runtime errors, is that the program displayed a nice error message and exited.

This most likely means that something in your program caused a call to the PLERROR macro, which called the `errormsg` function, which threw a PLearnException, which got caught in the very external try/catch of your main program, which printed it out (you main program *does* catch PLearn exceptions and report them, doesn't it???).

Tracing the problem is easy:

- Launch your favorite debugger (`gdb`) from within your favorite development environment (`emacs`).
- Put a breakpoint in the `errormsg` function by typing:
  `br 'PLearn::errormsg(`*TAB*
  Pressing the *TAB* key will complete the signature of the function for you. Note that the single quote at the beginning is important for this to work.

- `run` your program until it reaches the breakpoint.
- trace `up` the call stack and figure out what and why it happens.

Hints for using gdb:
`gdb` is always at quite a lag behind, playing catch-up with the latest compiler. Sot it *has* problems. Here are a few hints for working with it, or in spite of it...

- printing a `std::string` doesn't work. Cast it to a `char*` first, as in `p (char *) my_string`, or call `p my_string.c_str()`
- `gdb` often seems lost when you attempt to examine the insides of a complex object, replying that it can't find info on that class. In this case, unfortunately, you'll have to insert instructions in the code to print the desired debug info (with `cerr << ...`), recompile and rerun gdb.
- If you want to see an object on which you have a `PP` smart pointer (or similar type), you can access the raw pointer inside (it's called `ptr`!), for ex: `p my_var.ptr->value.length_`

I also suggest you learn using a good integrated development environment (IDE) like Emacs. Emacs has multiple windows, a compilation mode (pressing return on an error will bring you directly to the problematic line of the problematic file), and a gdb mode (with which you can easily follow the step by step execution, as an arrow is always displayed before the next instruction to be executed, and you can rapidly put a breakpoint (Ctrl-X SPACE) anywhere).

With the proper key definitions (learn how to define your own keys for maximum efficiency!), a complete recompile of your code, and reload in gdb is just one key away. You won't have to touch the rat. And yes, it even all works perfectly fine (multiple windows and all) inside a *single* text terminal over a telnet connexion for ex. (invoke it using `emacs -nw`).

7.4 Dirty runtime errors

By this, I mean `segmentation fault` and the like.

- First try running your code in `gdb`, as above.
- If this doesn't appear too helpful on its own (and probably won't), try running your program with `valgrind`. It's a great tool for catching memory bugs (like writing or reading from memory areas you never allocated or initialized.). You can run it like that, from a shell:
  `valgrind --gdb-attach=yes your_prg_and_its_args`

License

This document is covered by the license appearing after the title page.

The PLearn software library and tools described in this document are distributed under the following BSD-type license:

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. The name of the authors may not be used to endorse or promote
products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHORS ‘‘AS IS’’ AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN
NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```