```
#include "CMAC.h"
#include "LoggerDraw.h"

#define TILINGS_PER_GROUP 32

CMACBase::CMACBase( int numF, int numA, double r[], double m[], double res[] ):
  FunctionApproximator( numF, numA, r, m, res )
{
  minimumTrace = 0.01;
  numNonzeroTraces = 0;
  for ( int i = 0; i < RL_MEMORY_SIZE; i++ ) {
    weights[ i ] = 0;
    traces[ i ] = 0;
  }
  srand( (unsigned int) 0 );
  int tmp[ 2 ];
  float tmpf[ 2 ];
  colTab = new collision_table( RL_MEMORY_SIZE, 1 );

  GetTiles( tmp, 1, 1, tmpf, 0 );  // A dummy call to set the hashing table
}


void CMACBase::setState( double s[] )
{
  FunctionApproximator::setState( s );
loadTiles();
}

void CMACBase::updateWeights( double delta, double alpha )
{
  double tmp = delta * alpha / numTilings;
  for ( int i = 0; i < numNonzeroTraces; i++ ) {
    int f = nonzeroTraces[ i ];
    if ( f > RL_MEMORY_SIZE || f < 0 )
      cerr << "f is too big or too small!!" << f << endl;
    weights[ f ] += tmp * traces[ f ];
  }
}

// Decays all the (nonzero) traces by decay_rate, removing those below minimum_trace
void CMACBase::decayTraces( double decayRate )
{
  int f;
  for ( int loc = numNonzeroTraces - 1; loc >= 0; loc-- ) {
    f = nonzeroTraces[ loc ];
    if ( f > RL_MEMORY_SIZE || f < 0 )
      cerr << "DecayTraces: f out of range " << f << endl;
    traces[ f ] *= decayRate;
    if ( traces[ f ] < minimumTrace )
      clearExistentTrace( f, loc );
  }
}
```

```
// Clear any trace for feature f
void CMACBase::clearTrace( int f)
{
  if ( f > RL_MEMORY_SIZE || f < 0 )
    cerr << "ClearTrace: f out of range " << f << endl;
  if ( traces[ f ] != 0 )
    clearExistentTrace( f, nonzeroTracesInverse[ f ] );
}

// Set the trace for feature f to the given value, which must be positive
void CMACBase::setTrace( int f, double newTraceValue )
{
  if ( f > RL_MEMORY_SIZE || f < 0 )
    cerr << "SetTraces: f out of range " << f << endl;
  if ( traces[ f ] >= minimumTrace )
    traces[ f ] = newTraceValue;        // trace already exists
  else {
    while ( numNonzeroTraces >= RL_MAX_NONZERO_TRACES )
      increaseMinTrace(); // ensure room for new trace
    traces[ f ] = newTraceValue;
    nonzeroTraces[ numNonzeroTraces ] = f;
    nonzeroTracesInverse[ f ] = numNonzeroTraces;
    numNonzeroTraces++;
  }
}

// Set the trace for feature f to the given value, which must be positive
void CMACBase::updateTrace( int f, double deltaTraceValue )
{
  setTrace( f, traces[ f ] + deltaTraceValue );
}


void CMAC::loadTiles()
{
  int tilingsPerGroup = TILINGS_PER_GROUP;  /* num tilings per tiling group */
  numTilings = 0;

  /* These are the 'tiling groups'  --  play here with representations */
  /* One tiling for each state variable */
  for ( int v = 0; v < getNumFeatures(); v++ ) {
    for ( int a = 0; a < getNumActions(); a++ ) {
      GetTiles1( &(tiles[ a ][ numTilings ]), tilingsPerGroup, colTab,
                          state[ v ] / getResolution( v ), a , v );
    }
    numTilings += tilingsPerGroup;
  }
  if ( numTilings > RL_MAX_NUM_TILINGS )
    cerr << "TOO MANY TILINGS! " << numTilings << endl;
}
```

```
double CMAC::computeQ( int action )
{
  double q = 0;
  for ( int j = 0; j < numTilings; j++ ) {
    q += weights[ tiles[ action ][ j ] ];
  }
  return q;
}

void CMAC::clearTraces( int action )
{
  for ( int j = 0; j < numTilings; j++ )
    clearTrace( tiles[ action ][ j ] );
}

void CMAC::updateTraces( int action )
{
  for ( int j = 0; j < numTilings; j++ )      //replace/set traces F[a]
    setTrace( tiles[ action ][ j ], 1.0 );
}
```

```
CMAC_RBF::CMAC_RBF( int numF, int numA, double r[], double m[], double res[] ):
  CMACBase( numF, numA, r, m, res )
{
  RBF_spread = 4;
  RBF_sigma = 0.25; //*met 5/20/05 .5;
  RBF_tiles = (int)(RBF_spread * RBF_sigma + 0.5);
}

void CMAC_RBF::loadTiles()
{
  int tilingsPerGroup = TILINGS_PER_GROUP;  /* num tilings per tiling group */
  double tiling_displace = 1.0 / tilingsPerGroup;  // YL
  numTilings = 0;

  /* These are the 'tiling groups'  --  play here with representations */
  /* One tiling for each state variable */
  for ( int v = 0; v < getNumFeatures(); v++ ) {
    double nx = state[ v ] / getResolution( v );
    for ( int a = 0; a < getNumActions(); a++ ) {
      for ( int i = -RBF_tiles; i <= RBF_tiles; i++ ) {
        GetTiles1( &(tiles[ a ][ i + RBF_tiles ][ numTilings ]),
                          tilingsPerGroup, colTab, nx + i, a, v );

        for ( int k = 0; k < tilingsPerGroup; k++ ) {
          // the center for this tile
          double c = floor(nx) + 0.5 + k * tiling_displace;
          normalizedDistances[ a ][ i + RBF_tiles ][ numTilings + k ]
                = fabs( nx + i - c );
```

```
      }
     }
    }
    numTilings += tilingsPerGroup;
  }
  if ( numTilings > RL_MAX_NUM_TILINGS )
    cerr << "TOO MANY TILINGS! " << numTilings << endl;
}

//////////////////

double CMAC_RBF::computeQ( int action )
{
  double q = 0;
  // normalizedDistances are calculated in LoadTiles
  for ( int i = 0; i <= 2*RBF_tiles + 1; i++ ) {
    for ( int j = 0; j < numTilings; j++ ) {
      q += weights[ tiles[ action ][ i ][ j ] ]
             * rbf( normalizedDistances[ action ][ i ][ j ] );
    }
  }

  return q;
}
```

```
#include "NNet.h"
#include "LoggerDraw.h"

NNet::NNet( int numF, int numA, double r[], double m[], double res[] ):
  FunctionApproximator( numF, numA, r, m, res )
{
  numHiddens = 20;
  state[ getNumFeatures() ] = 1.0;

  for ( int a = 0; a < MAX_ACTIONS; ++a ) {
    for ( int i = 0; i < MAX_STATE_VARS; ++i ) {
      for ( int j = 0; j < MAX_HIDDENS; ++j ) {
        inWeights[ a ][ i ][ j ] = drand48();
        inTraces[ a ][ i ][ j ] = 0;
      }
    }
  }

  for ( int a = 0; a < MAX_ACTIONS; ++a ) {
    for ( int j = 0; j < MAX_HIDDENS; ++j ) {
      outWeights[ a ][ j ] = drand48();
      outTraces[ a ][ j ] = 0;
    }

    hiddenOutputs[ a ][ numHiddens ] = 1.0;
```

```
  }
}

double NNet::computeQ( int action )
{
  for ( int j = 0; j < numHiddens; ++j ) {
    hiddenOutputs[ action ][ j ] = 0;
    for (int i = 0; i < getNumFeatures(); ++i) {
      hiddenOutputs[ action ][ j ] +=
            ( ( state[ i ] - getMinValue( i ) ) / getRange( i ) - 0.5 )
            * inWeights[ action ][ i ][ j ];
    }
    hiddenOutputs[ action ][ j ] = sigm( hiddenOutputs[ action ][ j ] );
  }

  double result = 0;
  for ( int j = 0; j <= numHiddens; ++j ) {
    result += outWeights[ action ][ j ] * hiddenOutputs[ action ][ j ];
  }

  return result;
}


void NNet::updateWeights( double delta, double alpha )
{
  double tmp = delta * alpha / numHiddens;
  //printf("updateWeights( delta=%f alpha=%f numHiddens=%d tmp=%f\n", delta, alpha, numHiddens, tmp);
  for ( int action = 0; action < getNumActions(); ++action ) {
    for ( int i = 0; i < getNumFeatures(); ++i ) {
      for ( int j = 0; j < numHiddens; ++j ) {
        inWeights[ action ][ i ][ j ] += tmp * inTraces[ action ][ i ][ j ];
}
    }
  }

  for ( int action = 0; action < getNumActions(); ++action ) {
    for ( int j = 0; j <= numHiddens; ++j ) {
       outWeights[action][j], action, j, outTraces[action][j]);
       outWeights[ action ][ j ] += tmp * outTraces[ action ][ j ];
    }
  }
}
```

```
// Decays all the (nonzero) traces by decay_rate, removing those below minimum_trace
void NNet::decayTraces( double decayRate )
{
  for ( int action = 0; action < getNumActions(); ++action ) {
    for ( int i = 0; i < getNumFeatures(); ++i ) {
      for ( int j = 0; j < numHiddens; ++j ) {
        inTraces[ action ][ i ][ j ] *= decayRate;
      }
    }
  }

  for ( int action = 0; action < getNumActions(); ++action ) {
    for ( int j = 0; j <= numHiddens; ++j ) {
      outTraces[ action ][ j ] *= decayRate;
    }
  }
}

void NNet::clearTraces( int action )
{
  for ( int i = 0; i < getNumFeatures(); ++i ) {
    for ( int j = 0; j < numHiddens; ++j ) {
      inTraces[ action ][ i ][ j ] = 0;
    }
  }

  for ( int j = 0; j <= numHiddens; ++j ) {
    outTraces[ action ][ j ] = 0;
  }
}

void NNet::updateTraces( int action )
{
  for ( int i = 0; i < getNumFeatures(); ++i ) {
    for ( int j = 0; j < numHiddens; ++j ) {
      double gradient =
        outWeights[ action ][ j ] * hiddenOutputs[ action ][ j ] * ( 1 - hiddenOutputs[ action ][ j ] )
        * ( ( state[ i ] - getMinValue( i ) ) / getRange( i ) - 0.5 );
      inTraces[ action ][ i ][ j ] += gradient;
    }
  }

  for ( int j = 0; j <= numHiddens; ++j ) {
    double gradient = hiddenOutputs[ action ][ j ];
    outTraces[ action ][ j ] += gradient;
  }
}
```